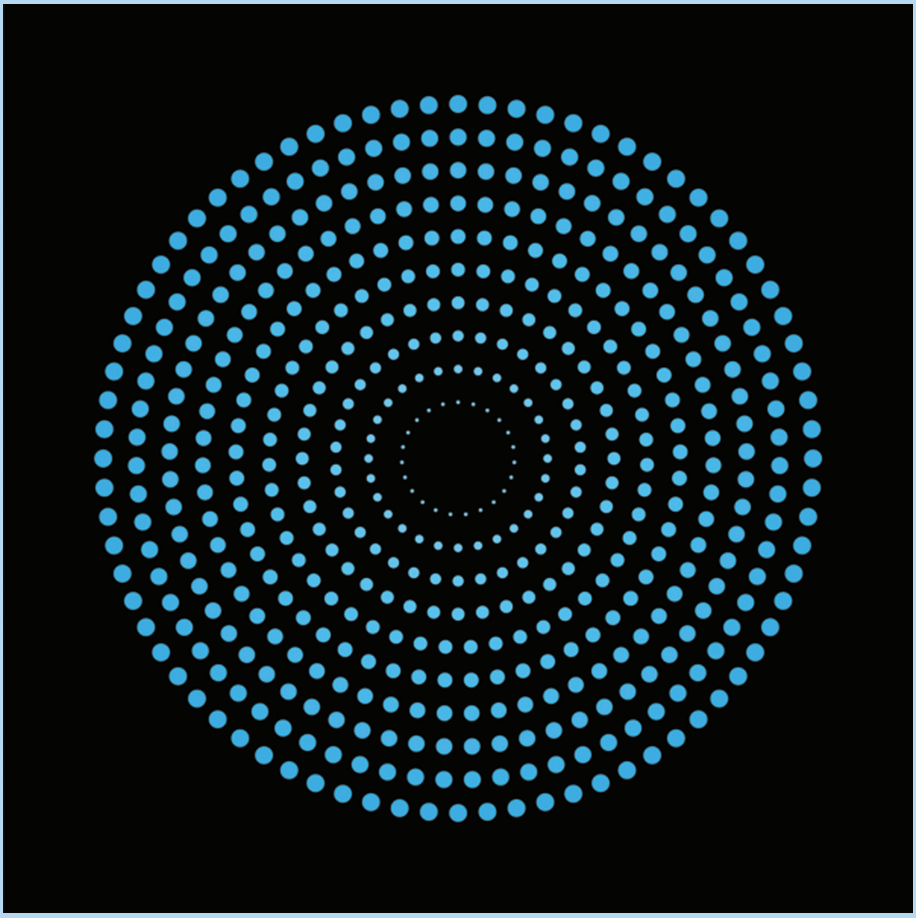
# Operating System Design

## The Xinu Approach

### Linksys Version



Douglas Comer

# Operating System Design

## The Xinu Approach

### Linksys Version

This page intentionally left blank

# Operating System Design

## The Xinu Approach

### Linksys Version

Douglas Comer

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group an **informa** business

A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742


Version Date: 20110629

International Standard Book Number-13: 978-1-4398-8111-8 (eBook - PDF)

*To my wife, Chris, and our children, Sharon and Scott*

This page intentionally left blank

# Contents

# Preface

Building a computer operating system is like weaving a fine tapestry. In each case, the ultimate goal is a large, complex artifact with a unified and pleasing design, and in each case, the artifact is constructed with small, intricate steps. As in a tapestry, small details are essential because a minor mismatch is easily noticed — like stitches in a tapestry, each new piece added to an operating system must fit the overall design. Therefore, the mechanics of assembling pieces forms only a small part of the overall process; a masterful creation must start with a pattern, and all artisans who work on the system must follow the pattern.

Ironically, few operating system textbooks or courses explain underlying patterns and principles that form the basis for operating system construction. Students form the impression that an operating system is a black box, and textbooks reenforce the misimpression by explaining operating system features and focusing on how to use operating system facilities. More important, because they only learn how an operating system appears from the outside, students are left with the feeling that an operating system consists of a set of interface functions that are connected by a morass of mysterious code containing many machine-dependent tricks.

Surprisingly, students often graduate with the impression that work on operating systems is over: existing operating systems, constructed by commercial companies and the open source community, suffice for all needs. Nothing could be further from the truth. Ironically, even though fewer companies are now producing conventional operating systems for personal computers, the demand for operating system expertise is rising and companies are hiring students to work on operating systems. The demand arises from inexpensive microprocessors embedded in devices such as smart phones, video games, iPods, Internet routers, cable and set-top boxes, and printers.

When working in the embedded world, knowledge of principles and structures is essential because a programmer may be asked to build new mechanisms inside an operating system or to modify an operating system for new hardware. Furthermore, writing applications for embedded devices requires an appreciation for the underlying operating system — it is impossible to exploit the power of small embedded processors without understanding the subtleties of operating system design.

This book removes the mystery from operating system design, and consolidates the body of material into a systematic discipline. It reviews the major system components, and imposes a hierarchical design paradigm that organizes the components in an orderly, understandable manner. Unlike texts that survey the field by presenting as many alternatives as possible, the reader is guided through the construction of a conventional process-based operating system, using practical, straightforward primitives. The text

begins with a bare machine, and proceeds step-by-step through the design and implementation of a small, elegant system. The system, called Xinu, serves as an example and a pattern for system design.

Although it is small enough to fit into the text, Xinu includes all the components that constitute an ordinary operating system: memory management, process management, process coordination and synchronization, interprocess communication, real-time clock management, device-independent I/O, device drivers, network protocols, and a file system. The components are carefully organized into a hierarchy of layers, making the interconnections among them clear and the design process easy to follow. Despite its size, Xinu retains much of the power of larger systems. Xinu is not a toy — it has been used in many commercial products by companies such as Mitsubishi, Lexmark, HP, IBM, and Woodward (woodward.com), Barnard Software, and Mantissa Corporation. An important lesson to be learned is that good system design can be as important on small embedded systems as on large systems and that much of the power arises from choosing good abstractions.

The book covers topics in the order a designer follows when building a system. Each chapter describes a component in the design hierarchy, and presents example software that illustrates the functions provided by that level of the hierarchy. The approach has several advantages. First, each chapter explains a successively larger subset of the operating system than the previous chapters, making it possible to think about the design and implementation of a given level independent of the implementation of succeeding levels. Second, the details of a chapter can be skipped on first reading — a reader only needs to understand the services that the level provides, not how those services are implemented. Third, reading the text sequentially allows a reader to understand a given function before the function is used to build others. Fourth, intellectually deep subjects like support for concurrency arise early, before higher-level operating system services have been introduced. Readers will see that the most essential functionality only occupies a few lines of code, which allows us to defer the bulk of the code (networking and file systems) until later when the reader is better prepared to understand details and references to basic functions.

Unlike many other books on operating systems, this text does not attempt to review every alternative for each system component, nor does it survey existing commercial systems. Instead, it shows the implementation details of one set of primitives, usually the most popular set. For example, the chapter on process coordination explains semaphores (the most widely accepted process coordination primitives), relegating a discussion of other primitives (e.g., monitors) to the exercises. Our goal is to remove all the mystery about how primitives can be implemented on conventional hardware. Once the essential magic of a particular set of primitives is understood, the implementation of alternative versions will be easy to master.

The example code in the text runs on a Linksys E2100L *Wireless Router*, which is marketed as a consumer product at retail stores. We do not use the Linksys hardware as a wireless router. Instead, we open the device, connect a serial line to the console port, use the serial line to interrupt the normal bootstrap process, and enter commands that

cause the hardware to download and run a copy of Xinu. In essence, we ignore the software that the vendor supplies, take over the underlying hardware, and run Xinu instead.

The book is designed for advanced undergraduate or graduate-level courses, and for computing professionals who want to understand operating systems. Although there is nothing inherently difficult about any topic, there is more than enough material for a semester course. Few undergraduates are adept at reading sequential programs, and fewer still understand the details of a run-time environment or machine architecture. Thus, they need to be guided through the chapters on process management and process synchronization carefully. If time is limited, I recommend covering Chapters 1–7 (process management), 9 (basic memory management), 12 (interrupt processing), 13 (clock management), 14 (device-independent I/O), and 19 (file systems). For a full semester undergraduate course, it is important for students to see basic remote access, such as the remote file system in Chapter 20. In a graduate course, students should read the entire book, and class discussion should focus on subtleties, tradeoffs, and comparison of alternatives. Two topics should be included in all classes: the change during initialization when a running program is transformed into a process and the transformation in the shell when a sequence of characters on an input line become string arguments passed to a command process.

In all cases, learning improves dramatically if students have hands-on experience with the system in a lab. Ideally, they can start to use the system in the first few days or weeks of the class before they try to understand the internal structure. Chapter 1 provides a few examples and encourages experimentation. (It is surprising how many students take operating system courses without ever writing a concurrent program or using system facilities.)

Covering most of the material in one semester demands an extremely rapid pace usually unattainable by undergraduates. Choosing items to omit depends largely on the background of students who take the course. In system courses, class time will be needed to help students understand the motivation as well as the details. If students have taken a data structures course that covers memory management and list manipulation, Chapters 4 and 9 can be skipped. If students will take a course in networking, Chapter 17 on network protocols can be skipped. The text includes chapters on both a remote disk system (18) and a remote file system (20); one of the two can be skipped. The chapter on a remote disk system may be slightly more pertinent because it introduces the topic of disk block caching, which is central in many operating systems.

In grad courses, class time can be spent discussing motivations, principles, tradeoffs, alternative sets of primitives, and alternative implementations. Students should emerge with a firm understanding of the process model and the relationship between interrupts and processes as well as the ability to understand, create, and modify system components. They should have a complete mental model of the entire system, and know how all the pieces interact.

Programming projects are strongly encouraged at all levels. Many exercises suggest modifying or measuring the code, or trying alternatives. The software is available

for free download, and a link to instructions for building a Linksys lab can also be found on the website:

www.xinu.cs.purdue.edu

Because the Linksys hardware is inexpensive, a lab can be constructed at very low cost. Alternatively, versions of the software are available for other hardware platforms, including the x86, and as of publication, a limited version for ARM.

Many of the exercises suggest improvements, experiments, and alternative implementations. Larger projects are also possible. Examples that have been used with various hardware include: a virtual memory system, mechanisms to synchronize execution across computers, and the design of a virtual network. Other students have transported Xinu to various processors or built device drivers for various I/O devices.

A background in basic programming is assumed. A reader should understand basic data structures, including linked lists, stacks, and queues, and should have written programs in C.

Furthermore, I encourage designers to code in high-level languages whenever possible, reverting to assembly language only when necessary. Following this approach, most of Xinu has been written in C. A few machine-dependent functions, such as the context switch and lowest levels of the interrupt dispatcher are written in assembly language. The explanations and comments accompanying the assembly code makes it possible to understand without learning assembly language in detail. Versions for other platforms are available, making it possible to compare the amount of code required on a MIPS processor to the code required for other processor architectures, such as an x86.

I owe much to my experiences, good and bad, with commercially available operating systems. Although Xinu differs internally from existing systems, the fundamental ideas are not new. Although many concepts and names have been taken from Unix, readers should be aware that many of the function arguments and the internal structure of the two systems differ dramatically — applications written for one system will not run on the other without modification.

I gratefully acknowledge the help of many people who contributed ideas, hard work, and enthusiasm to the Xinu project. Over the years, many graduate students at Purdue have worked on the system, ported it, and written device drivers. The version in this book represents a complete rewrite that started with the original system (now thirty years old), and retains the elegance of the original design. Dennis Brylow ported Xinu to the Linksys platform, and created many of the low-level pieces, including the startup code, context switch, and the Ethernet driver. Dennis also designed the rebooter mechanism we are using in the lab at Purdue. Special thanks go to my wife and partner, Christine, whose careful editing and suggestions made many improvements throughout.

Douglas E. Comer

August, 2011

# About The Author

**Douglas Comer**, Distinguished Professor of Computer Science at Purdue University, is an internationally recognized expert on computer networking, the TCP/IP protocols, the Internet, and operating systems design. The author of numerous refereed articles and technical books, he is a pioneer in the development of curriculum and laboratories for research and education.

A prolific author, Dr. Comer's popular books have been translated into sixteen languages, and are used in industry as well as computer science, engineering, and business departments around the world. His landmark three-volume series *Internetworking With TCP/IP* revolutionized networking and network education. His textbooks and innovative laboratory manuals have shaped and continue to shape graduate and undergraduate curricula.

The accuracy and insight of Dr. Comer's books reflect his extensive background in computer systems. His research spans both hardware and software. He has created Xinu, a complete operating system, written device drivers, and implemented network protocol software for conventional computers as well as network processors. Software that has resulted from Dr. Comer's research has been used by industry in a variety of products.

Dr. Comer has created and teaches courses on network protocols, operating systems, and computer architecture for a variety of audiences, including courses for engineers as well as academic audiences. His innovative educational laboratories allow him and his students to design and implement working prototypes of large, complex systems, and measure the performance of the resulting prototypes. He continues to teach at companies, universities, and conferences around the world. In addition, Dr. Comer consults for industry on the design of computer networks and systems.

For twenty years, Professor Comer served as editor-in-chief of the research journal *Software — Practice and Experience*. While on an extended leave from Purdue, he served as Vice President of Research at Cisco Systems. He is a Fellow of the ACM, a Fellow of the Purdue Teaching Academy, and a recipient of numerous awards, including a Usenix Lifetime Achievement award.

Additional information about Dr. Comer can be found at:

www.cs.purdue.edu/people/comer

and information about his books can be found at:

www.comerbooks.com

# Chapter Contents

# 1

# Introduction and Overview

*Our little systems have their day.*

— Alfred, Lord Tennyson

## 1.1 Operating Systems

Hidden in every intelligent device and computer system is the software that controls processing, manages resources, and communicates with devices such as display screens, computer networks, disks, and printers. Collectively, the code that performs control and coordination chores has been referred to as an *executive*, a *monitor*, a *task manager*, or a *kernel*; we will use the broader term *operating system*.

Computer operating systems are among the most complex objects created by mankind: they allow multiple computational processes and users to share a CPU simultaneously, protect data from unauthorized access, and keep independent input/output (I/O) devices operating correctly. The high-level services an operating system offers are all achieved by issuing detailed commands to intricate hardware. Interestingly, an operating system is not an independent mechanism that controls a computer from the outside — it consists of software that is executed by the same processor that executes applications. In fact, when a processor is executing an application, the processor cannot be executing the operating system and vice versa.

Arranging mechanisms that guarantee an operating system will always regain control after an application runs complicates system design. The most impressive aspect of an operating system, however, arises from the difference in functionality between services and hardware: an operating system provides high-level services over extremely low-level hardware. As the book proceeds, we will understand how crude the underlying hardware can be, and see how much system software is required to handle even a simple device such as a serial interface. The philosophy is straightforward: an operating

system should provide abstractions that make programming easier rather than abstractions that reflect the underlying hardware. Thus, we conclude:

> *An operating system is designed to hide low-level hardware details and to create an abstract machine that provides applications with high-level services.*

Operating system design is not a well-known craft. In the beginning, because computers were scarce and expensive, only a few programmers had an opportunity to work on operating systems. Now that advances in micro-electronic technology have reduced fabrication costs and made microprocessors inexpensive, operating systems are commodities, and few programmers work on them. Interestingly, microprocessors have become so inexpensive that most electronic devices are now constructed from programmable processors rather than from discrete logic. As a result, designing and implementing software systems for microcomputers and microcontrollers is no longer a task reserved for a few specialists; it has become a skill expected of competent systems programmers.

Fortunately, our understanding of operating systems has grown along with the technology used to produce new machines. Researchers have explored fundamental issues, formulated design principles, identified essential components, and devised ways that components can work together. More important, researchers have identified abstractions, such as files and current processes, that are common to all operating systems, and have found efficient implementations for the abstractions. Finally, we have learned how to organize the components of an operating system into a meaningful structure that simplifies system design and implementation.

Compared to its early counterparts, a modern system is simple, clean, and portable. A well-designed system follows a basic pattern that partitions software into a set of basic components. As a result, a modern system can be easier to understand and modify, can contain less code, and has less processing overhead than early systems.

Vendors that sell large commercial operating systems include many extra software components along with an operating system. For example, a typical software distribution includes compilers, linkers, loaders, library functions, and a set of applications. To distinguish between the extras and the basic system, we sometimes use the term *kernel* to refer to the code that remains resident in memory and provides key services such as support for concurrent processes. Throughout the text, we will assume the term *operating system* refers to the kernel, and does not include all additional facilities. A design that minimizes the facilities in a kernel is sometimes called a *microkernel* design; our discussions will concentrate on a microkernel.

## 1.2 Approach Used In The Text

This book is a guide to the structure, design, and implementation of operating system kernels. Instead of merely listing operating system features and describing systems abstractly, the book takes an engineering approach. It shows how each abstraction can be built and how the abstractions can be organized into an elegant, efficient design.

Our approach provides two advantages. First, because the text covers every part of the system, a reader will see how an entire system fits together, not merely how one or two parts interact. Second, because source code is available for all pieces described in the text, no mystery remains about any part of the implementation — a reader can obtain a copy of the system to examine, modify, instrument, measure, extend, or transport to another architecture. By the end of the book, a reader will see how each piece of an operating system fits into the design, and will be prepared to understand alternative design choices.

Our focus on implementation means that the software forms an integral part of the text. In fact, the code provides a centerpiece for discussion; one must read and study the program listings to appreciate the underlying subtlety and engineering detail. The example code is minimal, which means a reader can concentrate on concepts without wading through many pages of code. Some of the exercises suggest improvements or modifications that require a reader to delve into details or invent alternatives; a skillful programmer will find additional ways to improve and extend the system.

## 1.3 A Hierarchical Design

If designed well, the interior of an operating system can be as elegant and clean as the best conventional program. The design described in this book achieves elegance by partitioning system functions into eight major categories, and organizing the components into a multi-level hierarchy. Each level of the system provides an abstract service, implemented in terms of the abstract services provided by lower levels. The approach offers a property that will become apparent: successively larger subsets of the levels can be selected to form successively more powerful systems. We will see how a hierarchical approach provides a model for designers that helps reduce complexity.

Another important property of our approach arises from run-time efficiency — a designer can structure pieces of an operating system into a hierarchy without introducing extra overhead. In particular, our approach differs from a conventional layered system in which a function at level $K$ can only invoke functions at level $K-1$. In our multi-level approach, the hierarchy only provides a conceptual model for a designer — at runtime, a function at a given level of the hierarchy can invoke any of the functions in lower levels directly. We will see that direct invocation makes the entire system efficient.

Figure 1.1 illustrates the hierarchy used in the text, gives a preview of the components we will discuss, and shows the structure into which all pieces are organized.

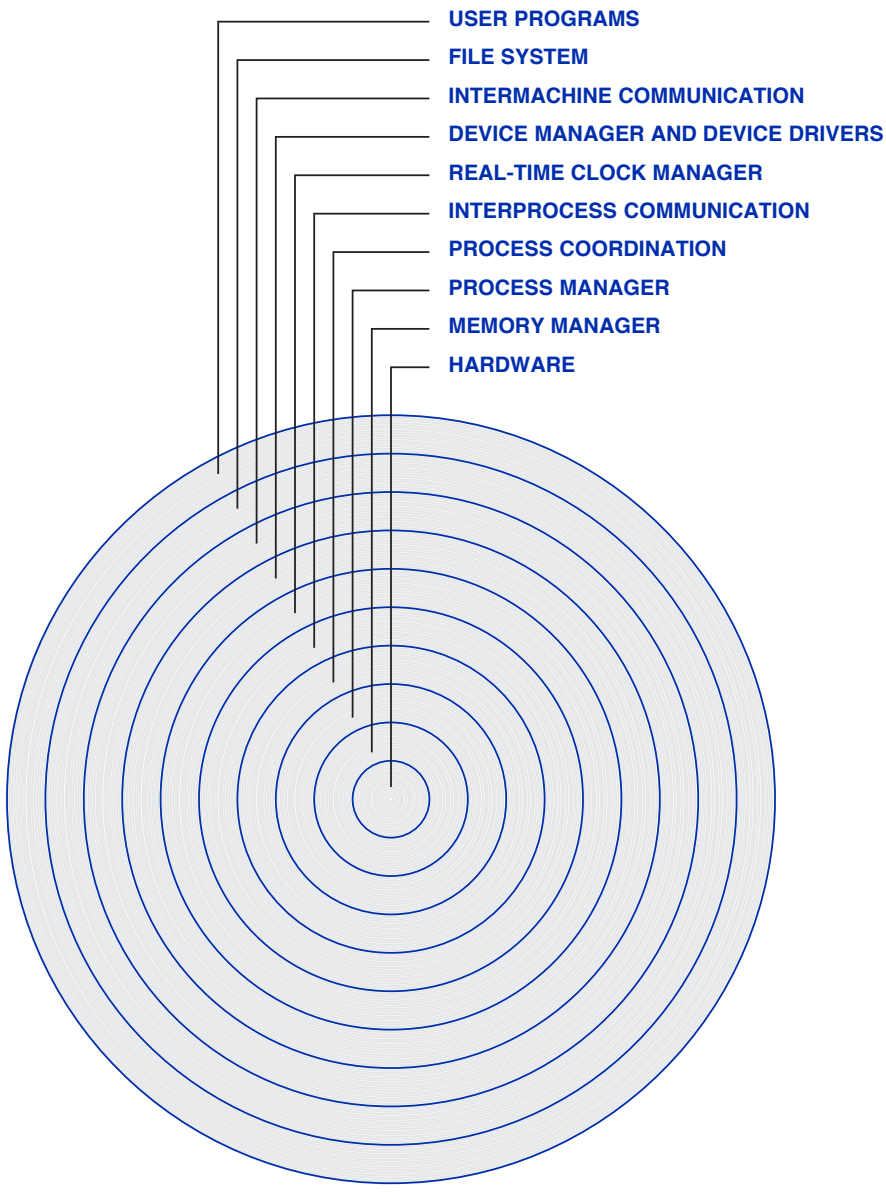



**Figure 1.1** The multi-level organization used in the text.

At the heart of the hierarchy lies the computer hardware. Although not part of the operating system itself, modern hardware includes features that allow tight integration with an operating system. Thus, we think of the hardware as forming level zero of our hierarchy.

Building out from the hardware, each higher level of operating system software provides more powerful primitives that shield applications from the raw hardware. A

memory manager controls and allocates memory. Process management forms the most fundamental component of the operating system, and includes a scheduler and context switch. Functions in the next level constitute the rest of the process manager, providing primitives to create, kill, suspend, and resume processes. Just beyond the process manager comes a process coordination component that implements semaphores. Functions for real-time clock management occupy the next level, and allow application software to delay for a specified time. On top of the real-time clock level lies a level of device-independent I/O routines that provide familiar services, such as *read* and *write*. Above the device routines, a level implements network communication, and the level above that implements a file system.

The internal organization of a system should not be confused with the services the system provides. Although components are organized into levels to make the design and implementation cleaner, the resulting hierarchical structure does not restrict system calls at run-time. That is, once the system has been built, facilities from all levels of the hierarchy can be exposed to applications. For example, an application can invoke semaphore functions, such as *wait* and *signal*, that reside in the process coordination level just as easily as it can invoke functions such as *putc* that reside in an outer level. Thus, the multi-level structure describes only the internal implementation, and does not restrict the services the system provides.

## 1.4 The Xinu Operating System

Examples in the book are taken from the *Xinu*† operating system. Xinu is a small, elegant system that is intended for use in an embedded environment, such as a cell phone or an MP3 player. Typically, Xinu is loaded into memory along with a fixed set of applications when the system boots. Of course, if memory is constrained or the hardware architecture uses a separate memory for instructions, Xinu can be executed from flash or another read-only memory. In a typical system, however, executing from main memory produces higher performance.

Xinu is not a toy; it is a powerful operating system that has been used in commercial products. For example, Xinu was used in pinball games sold under the Williams/Bally brand (the major manufacturer), Woodward Corporation uses Xinu to control large gas/steam and diesel/steam turbine engines, and Lexmark Corporation used Xinu as the operating system in many of its printers. In each case, when the device was powered on, the hardware loaded a memory image that contained Xinu.

Xinu contains the fundamental components of an operating system, including: process, memory, and timer management mechanisms, interprocess communication facilities, device-independent I/O functions, and Internet protocol software. Xinu can control I/O devices and perform chores such as reading keystrokes from a keyboard or keypad, displaying characters on an output device, managing multiple, simultaneous computations, controlling timers, passing messages between computations, and allowing applications to access the Internet.

†The name stands for Xinu Is Not Unix. As we will see, the internal structure of Xinu differs completely from the internal structure of Unix (or Linux). Xinu is smaller, more elegant, and easier to understand.

Xinu illustrates how the hierarchical design that is described above applies in practice. It also shows how all the pieces of an operating system function as a uniform, integrated whole, and how an operating system makes services available to application programs.

## 1.5 What An Operating System Is Not

Before proceeding into the design of an operating system, we should agree on what we are about to study. Surprisingly, many programmers do not have a correct intuitive definition of an operating system. Perhaps the problem arises because vendors and computer professionals often apply the terminology broadly to refer to all software supplied by a vendor as well as the operating system itself, or perhaps confusion arises because few programmers access system services directly. In any case, we can clarify the definition quickly by ruling out well-known items that are not part of the operating system kernel.

First, an operating system is not a language or a compiler. Of course, an operating system must be written in some language, and languages have been designed that incorporate operating systems features and facilities. Further confusion arises because a software vendor may offer one or more compilers that have been integrated with their operating system. However, an operating system does not depend on an integrated language facility — we will see that a system can be constructed using a conventional language and a conventional compiler.

Second, an operating system is not a windowing system or a browser. Many computers and electronic devices have a screen that is capable of displaying graphics, and sophisticated systems permit applications to create and control multiple, independent windows. Although windowing mechanisms rely on an operating system, a windowing system can be replaced without replacing the operating system.

Third, an operating system is not a command interpreter. Embedded systems often include a *Command Line Interface* (*CLI*); some embedded systems rely on a CLI for all control. In a modern operating system, however, the command interpreter operates as an application program, and the interpreter can be changed without modifying the underlying system.

Fourth, an operating system is not a library of functions or methods. Application programs that send email, process documents, provide database access, or communicate over the Internet all use library routines, and the software found in libraries can offer significant functionality. Although many library routines use operating system services, the underlying operating system remains independent of the libraries.

Fifth, an operating system is not the first code that runs after a computer is powered on. Instead, the computer contains *firmware* (i.e., a program in non-volatile memory) that initializes various pieces of hardware, loads a copy of the operating system into memory, and then jumps to the beginning of the operating system. On a PC, for example, the firmware is known as the *Basic Input Output System* (*BIOS*).

## 1.6 An Operating System Viewed From The Outside

The essence of an operating system lies in the services it provides to applications. An application accesses operating system services by making *system calls*. In source code, a system call appears to be a conventional function invocation. However, when invoked at run-time, a system call transfers control to the operating system, which performs the requested service for the application. Taken as a set, system calls establish a well-defined boundary between applications and the underlying operating system that is called an *Application Program Interface* (*API*). The API defines the services that the system provides as well as the details of how an application uses the services.

To appreciate the interior of an operating system, one must first understand the characteristics of the API and see how applications use the services. This chapter introduces a few fundamental services, using examples from the Xinu operating system to illustrate the concepts. For example, the Xinu procedure *putc* writes a single character to a specified I/O device. *Putc* takes two arguments: a device identifier and a character to write. File *ex1.c* contains an example C program that writes the message "hi" on the console when run under Xinu:

```
/* ex1.c - main */

#include <xinu.h>

/*------------------------------------------------------------------------
 * main  --  write "hi" on the console
 *------------------------------------------------------------------------
 */
void    main(void)
{
        putc(CONSOLE, 'h'); putc(CONSOLE, 'i');
        putc(CONSOLE, '\r'); putc(CONSOLE, '\n');
}
```

The code introduces several conventions used throughout Xinu. The statement *#include <xinu.h>* inserts a set of declarations in a source program that allows the program to reference operating system parameters. For example, the Xinu configuration file defines symbolic constant *CONSOLE* to correspond to a console serial device a programmer uses to interact with the embedded system. Later we will see that *xinu.h* includes other include files, and we will learn how names like *CONSOLE* become synonymous with devices; for now, it is sufficient to know that the *include* statement must appear in any Xinu application.

To permit communication with an embedded system (e.g., for debugging), the serial device on the embedded system can be connected to a *terminal application* on a conventional computer. Each time a user presses a key on the computer's keyboard, the

terminal application sends the keystroke over the serial line to the embedded system. Similarly, each time the embedded system sends a character to the serial device, the terminal application displays the character on the user's screen. Thus, a console provides two-way communication between the embedded system and the outside world.

The main program listed above writes four characters to the console serial device: "h", "i", a carriage return, and a line feed. The latter two are control characters that move the cursor to the beginning of the next line. Xinu does not perform any special operations when the program sends control characters — control characters are merely passed on to the serial device just like alphanumeric characters. Control characters have been included in the example to illustrate that *putc* is not line-oriented; in Xinu, a programmer is responsible for terminating a line.

The example source file introduces two important conventions followed throughout the book. First, the file begins with a one-line comment that contains the name of the file (*ex1.c*). If a source file contains multiple functions, the name of each appears on the comment line. Knowing the names of files will help you locate them in a machine-readable copy of Xinu. Second, the file contains a block comment that identifies the start of a procedure (*main*). Having a block comment before each procedure makes it easy to locate functions in a given file.

## 1.7 Remainder Of The Text

The remainder of the text proceeds through the design of a system that follows the multi-level organization that Figure 1.1 illustrates. Chapter 2 describes concurrent programming and the services an operating system supplies. Successive chapters consider the levels in roughly the same order as they are designed and built: from the innermost outward. Each chapter explains the role of one level in the system, describes new abstractions, and illustrates the details with source code. Taken together, the chapters describe a complete, working system and explain how the components fit together in a clean and elegant design.

Although the bottom-up approach may seem awkward at first, it shows how an operating system designer builds a system. The overall structure of the system will start to become clear by Chapter 9. By the end of Chapter 15, readers will understand a minimal kernel capable of supporting concurrent programs. By Chapter 20, the system will include remote file access, and by Chapter 23, the design will include a complete set of operating system functions.

## 1.8 Perspective

Why study operating systems? It may seem pointless because commercial systems are widely available and relatively few programmers write operating system code. However, a strong motivation exists: even in small embedded systems, applications run on top of an operating system and use the services it provides. Therefore, understand-

ing how an operating system works internally helps a programmer appreciate concurrent processing and make sensible choices about system services.

## 1.9 Summary

An operating system is not a language, compiler, windowing system, browser, command interpreter, or library of procedures. Because most applications use operating system services, programmers need to understand operating system principles. Programmers who work on programmable electronic devices need to understand operating system design.

The text takes a practical approach. Instead of describing components and features of an operating system abstractly, an example system, Xinu, is used to illustrate concepts. Although it is small and elegant, Xinu is not a toy — it has been used in commercial products. Xinu follows a multi-level design in which software components are organized into eight conceptual levels. The text explains one level of the system at a time, beginning with the raw hardware and ending with a working operating system.

## EXERCISES

**1.1** Should an operating system make hardware facilities available to application programs? Why or why not?

**1.2** What are the advantages of using a real operating system in examples?

**1.3** What are the eight major components of an operating system?

**1.4** In the Xinu multi-level hierarchy, can a file system function depend on a process manager function? Can a process manager function depend on a file system function? Explain.

**1.5** Explore the system calls available on your favorite operating system, and write a program that uses them.

**1.6** Various programming languages have been designed that incorporate OS concepts such as processes and process synchronization primitives. Find an example language, and make a list of the facilities it offers.

**1.7** Search the web, and make a list of the major commercial operating systems that are in use.

**1.8** Compare the facilities in Linux and Microsoft's Windows operating systems. Does either one support functionality that is not available in the other?

**1.9** The set of functions that an operating system makes available to application programs is known as the *Application Program Interface* or the *system call interface*. Choose two example operating systems, count the functions in the interface that each makes available, and compare the counts.

**1.10** Extend the previous exercise by identifying functions that are available in one system but not in the other. Characterize the purpose and importance of the functions.

**1.11** How large is an operating system? Choose an example system, and find how many lines of source code are used for the kernel.

# Chapter Contents

# 2

# Concurrent Execution And Operating System Services

*From an article on a new operating system for the IBM PC: Real concurrency — in which one program actually continues to function while you call up and use another — is more amazing but of small use to the average person. How many programs do you have that take more than a few seconds to perform any task?*

– *New York Times,* 25 April 1989

## 2.1 Introduction

This chapter considers the concurrent programming environment that an operating system provides for applications. It describes a model of concurrent execution, and shows why applications that operate concurrently need mechanisms to coordinate and synchronize. It introduces basic concepts, such as processes and semaphores, and explains how applications use such facilities.

Instead of describing operating systems abstractly, the chapter uses concrete examples from the Xinu system to illustrate concepts such as concurrency and synchronization. The chapter contains trivial applications that capture the essence of concurrent execution in a few lines of code. Later chapters expand the discussion by explaining in detail how an operating system implements each of the facilities described.

## 2.2 Programming Models For Multiple Activities

Even small computing devices are designed to handle multiple tasks at the same time. For example, while a voice call is connected, a cell phone can display the time of day, listen for other incoming calls, and allow the user to adjust the volume. More complex computing systems allow a user to run multiple applications that execute at the same time. The question arises: how should the software in such systems be organized? Three basic approaches can be used:

- Synchronous event loop
- Asynchronous event handlers
- Concurrent execution

*Synchronous event loop*. The term *synchronous* refers to events that are coordinated. A *synchronous event loop* uses a single, large iteration to handle coordination. During a given iteration of the loop, the code checks each possible activity and invokes the appropriate handler. Thus, the code has a structure similar to the following:

```
while (1) {   /* synchronous loop runs forever */
        Update time-of-day clock;
        if (screen timeout has expired) {
                turn off the screen;
        }
        if (volume button is being pushed) {
                adjust volume;
        }
        if (text message has arrived) {
                Display notification for user;
        }
        ...
}
```

*Asynchronous event handlers*. A second alternative is used in systems where the hardware can be configured to invoke a *handler* for each event. For example, the code to adjust volume might be placed in memory at location 100, and the hardware is configured so that when the *volume* button is pressed, control transfers to location 100. Similarly, the hardware can be configured so that when a text message arrives, control transfers to location 200, and so on. A programmer writes a separate piece of code for each event, and uses global variables to coordinate their interactions. For example, if a user presses the *mute* button, the code associated with the mute event turns off the audio and records the status in a global variable. Later, when the user adjusts the volume, code associated with the volume button checks the global variable, turns on the audio, and changes the global variable to indicate that audio is on.

*Concurrent execution*. The third architecture used to organize multiple activities is the most significant: software is organized as a set of programs that each operate concurrently. The model is sometimes called *run-to-completion* because each computation appears to run until it chooses to stop. From a programmer's point of view, concurrent execution is a delight. Compared to synchronous or asynchronous events, concurrent execution is more powerful, easier to understand, and less error-prone.

The next sections describe operating systems that provide the support needed for concurrency, and characterize the concurrent model. Later chapters examine the underlying operating system mechanisms and functions that enable a concurrent programming model.

## 2.3 Operating System Services

What are the main services that an operating system supplies? Although the details vary from system to system, most systems supply the same basic services. The services (with the chapters of the text that describe them) are:

- Support for concurrent execution (5, 6)
- Facilities for process synchronization (7)
- Inter-process communication mechanisms (8)
- Protection among running applications (9, 10)
- Management of address spaces and virtual memory (10)
- High-level interface for I/O devices (12–14)
- Network communication (17)
- A file system and file access facilities (19–21)

Concurrent execution is at the heart of an operating system, and we will see that concurrency affects each piece of operating system code. Thus, we begin by examining the facilities an operating system offers for concurrency, and use concurrency to show how an application program invokes services.

## 2.4 Concurrent Processing Concepts And Terminology

Conventional programs are called *sequential* because a programmer imagines a computer executing the code statement by statement; at any instant, the machine is executing exactly one statement. Operating systems support an extended view of computation called *concurrent processing*. Concurrent processing means that multiple computations can proceed "at the same time."

Many questions arise about concurrent processing. It is easy to imagine $N$ independent programs being executed simultaneously by $N$ processors or $N$ cores, but it is difficult to imagine a set of independent computations proceeding simultaneously on a

computer that has fewer than *N* processing units. Is concurrent computation possible even if a computer has a single core? If multiple computations each proceed simultaneously, how does the system keep one program from interfering with others? How do the programs cooperate so that only one takes control of an input or output device at a given time?

Although many CPUs do incorporate some amount of parallelism, the most visible form of concurrency, multiple independent applications that execute simultaneously, is a grand illusion. To create the illusion, an operating system uses a technique, called *multiprogramming* — the operating system switches the available processor(s) among multiple programs, allowing a processor to execute one program for only a few milliseconds before moving on to another. When viewed by a human, the programs appear to proceed concurrently. Multiprogramming forms the basis of most operating systems. The only exceptions are a few systems that operate basic devices such as a simplistic remote control used with a television and safety-critical embedded systems, such as flight avionics and medical device controllers, that use a synchronous event loop to guarantee that tight time constraints can be met absolutely.

Systems that support multiprogramming can be divided into two broad categories:

- Timesharing
- Real-time

*Timesharing*. A *timesharing system* gives equal priority to all computations, and permits computations to start or terminate at any time. Because they allow computations to be created dynamically, timesharing systems are popular for computers that human users operate. A timesharing system allows a human to leave an email application running while using a browser to view a web page and to leave a background application playing music. The chief characteristic of a timesharing system is that the amount of processing a computation receives is inversely proportional to the load on the system — if *N* computations are executing, each computation receives approximately $1/N$ of the available CPU cycles. Thus, as more computations appear, each proceeds at a slower rate.

*Real-time*. Because it is designed to meet performance constraints, a *real-time system* does not treat all computations equally. Instead, a real-time system assigns priorities to computations, and schedules the processor carefully to insure that each computation meets its required schedule. The chief characteristic of a real-time system arises from its ability to give the CPU to high-priority tasks, even if other tasks are waiting. For example, by giving priority to voice transmission, a real-time system in a cell phone can guarantee that the conversation is uninterrupted, even if a user runs an application to view the weather or an application to play a game.

Designers of multiprogramming systems have used a variety of terms to describe a single computation, including *process*, *task*, *job*, and *thread of control*. The terms *process* or *job* often connote a single computation that is self-contained and isolated from other computations. Typically, a process occupies a separate region of memory, and the operating system prevents a process from accessing the memory that has been assigned to another process. The term *task* refers to a process that is declared statically. That is,

a programming language allows a programmer to declare a process similar to the way one declares a function. The term *thread* refers to a type of process that shares an address space with other threads. Shared memory means that members of the set can exchange information efficiently. Early scientific literature used the term *process* to refer to concurrent execution in a generic sense; the Unix operating system popularized the idea that each process occupied a separate address space. The Mach system introduced a two-level concurrent programming scheme in which the operating system allows a user to create one or more processes that each operate in an independent region of memory, and further allows a user to create multiple threads of control within a given process. Linux follows the Mach model. To refer to a Linux-style process, the word *Process* is written with an uppercase *P*.

Because it is designed for an embedded environment, Xinu permits processes to share an address space. To be precise, we might say that Xinu processes follow a *thread* model. However, because the term *process* is widely accepted, we will use it throughout the text to refer generically to a concurrent computation.

The next section helps distinguish concurrent execution from sequential execution by examining a few applications. As we will see, the difference plays a central role in operating system design — each piece of an operating system must be built to support concurrent execution.

## 2.5 Distinction Between Sequential And Concurrent Programs

When a programmer creates a conventional (sequential) program, the programmer imagines a single processor executing the program step-by-step without interruption or interference. When writing code for a concurrent program, however, a programmer must take a different view and imagine multiple computations executing simultaneously. The code inside an operating system provides an excellent example of code that must accommodate concurrency. At any given instant, multiple processes may be executing. In the simplest case, each process executes application code that no other process is executing. However, an operating system designer must plan for a situation in which multiple processes have invoked a single operating system function, or even a case where multiple processes are executing the same instruction. To further complicate matters, the operating system may switch the processor among processes at any time; no guarantee can be made about the relative speed of computation in a multiprogramming system.

Designing code to operate correctly in a concurrent environment provides a tough intellectual challenge because a programmer must insure that all processes cooperate, no matter what operating system code they execute or in which order they execute. We will see how the notion of concurrent execution affects each line of code in an operating system.

To understand applications in a concurrent environment, consider the Xinu model. When it boots, Xinu creates a single process of execution that starts running the main program. The initial process can continue execution by itself, or it can create new processes. When a new process is created, the original process continues to execute,

and the new process executes concurrently. Either the original process or a new process can create additional processes that execute concurrently.

For example, consider the code for a concurrent application that creates two processes. Each process sends characters over the console serial device: the first process sends the letter *A*, and the second sends the letter *B*. File *ex2.c* contains the source code, which consists of a main program and two functions, *sndA* and *sndB*.

```
/* ex2.c - main, sndA, sndB */

#include <xinu.h>

void    sndA(void), sndB(void);

/*------------------------------------------------------------------------
 * main  --  example of creating processes in Xinu
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndA, 1024, 20, "process 1", 0) );
        resume( create(sndB, 1024, 20, "process 2", 0) );
}

/*------------------------------------------------------------------------
 * sndA  --  repeatedly emit 'A' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndA(void)
{
        while( 1 )
                putc(CONSOLE, 'A');
}

/*------------------------------------------------------------------------
 * sndB  --  repeatedly emit 'B' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndB(void)
{
        while( 1 )
                putc(CONSOLE, 'B');
}
```

In the code, the main program never calls either function directly. Instead, the main program calls two operating system functions, *create* and *resume*. Each call to *create* forms a new process that will begin executing instructions at the address specified by its first argument. In the example, the first call to *create* passes the address of function *sndA*, and the second call passes the address of function *sndB*.† Thus, the code creates a process to execute *sndA* and a process to execute *sndB*. *Create* establishes a process, leaves the process ready to execute but temporarily suspended, and returns an integer value that is known as a *process identifier* or *process ID*. The operating system uses the process ID to identify the newly created process; an application uses the process ID to reference the process. In the example, the main program passes the ID returned by *create* to *resume* as an argument. *Resume* starts (unsuspends) the process, allowing the process to begin execution. The distinction between normal function calls and process creation is:

> *A normal function call does not return until the called function completes. Process creation functions* create *and* resume *return to the caller immediately after starting a new process, which allows execution of both the existing process and the new process to proceed concurrently.*

In Xinu, all processes execute concurrently. That is, execution of a given process continues independent of other processes unless a programmer explicitly controls interactions among processes. In the example, the first new process executes code in function *sndA*, sending the letter *A* continuously, and the second executes code in function *sndB*, sending the letter *B* continuously. Because the processes execute concurrently, the output is a mixture of *A*s and *B*s.

What happens to the main program? Remember that in an operating system, each computation corresponds to a process. Therefore, we should ask, "What happens to the process executing the main program?" Because it has reached the end of the main program, the process executing the main program exits after the second call to *resume*. Its exit does not affect the newly created processes — they continue to send *A*s and *B*s. A later section describes process termination in more detail.

## 2.6 Multiple Processes Sharing A Single Piece Of Code

The example in file *ex2.c* shows each process executing a separate function. It is possible, however, for multiple processes to execute the same function. Arranging for processes to share code can be essential in an embedded system that has a small memory. To see an example of processes sharing code, consider the program in file *ex3.c*.

---

†Other arguments to *create* specify the stack space needed, a scheduling priority, a name for the process, the count of arguments passed to the process, and (when applicable) the argument values passed to the process; we will see details later.

```
/* ex3.c - main, sndch */

#include <xinu.h>

void    sndch(char);

/*------------------------------------------------------------------------
 * main  --  example of 2 processes executing the same code concurrently
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndch, 1024, 20, "send A", 1, 'A') );
        resume( create(sndch, 1024, 20, "send B", 1, 'B') );
}

/*------------------------------------------------------------------------
 * sndch  --  output a character on a serial device indefinitely
 *------------------------------------------------------------------------
 */
void    sndch(
          char  ch                      /* character to emit continuously */
        )
{
        while ( 1 )
                putc(CONSOLE, ch);
}
```

As in the previous example, a single process begins executing the main program. The process calls *create* twice to start two new processes that each execute code from function *sndch*. The final two arguments in the call to *create* specify that *create* will pass one argument to the newly created process and a value for the argument. Thus, the first process receives the character *A* as an argument, and the second process receives character *B*.

Although they execute the same code, the two processes proceed concurrently without any effect on one another. In particular, each process has its own copy of arguments and local variables. Thus, one process emits *A*s, while the other process emits *B*s. The key point is:

> *A program consists of code executed by a single process of control. In contrast, concurrent processes are not uniquely associated with a piece of code; multiple processes can execute the same code simultaneously.*

The examples provide a hint of the difficulty involved in designing an operating system. Not only must each piece be designed to operate correctly by itself, the designer must also guarantee that multiple processes can execute a given piece of code concurrently without interfering with one another.

Although processes can share code and global variables, each process must have a private copy of local variables. To understand why, consider the chaos that would result if all processes shared every variable. If two processes tried to use a shared variable as the index of a *for* loop, for example, one process might change the value while another process was in the midst of executing the loop. To avoid such interference, the operating system creates an independent set of local variables for each process.

Function *create* also allocates an independent set of arguments for each process, as the example in file *ex3.c* demonstrates. In the calls to *create*, the last two arguments specify a count of values that follow (1 in the example), and the value that the operating system passes to the newly created process. In the code, the first new process has character *A* as an argument, and the process begins execution with formal parameter *ch* set to *A*. The second new process begins with *ch* set to *B*. Thus, the output contains a mixture of both letters. The example points out a significant difference between the sequential and concurrent programming models.

*Storage for local variables, function arguments, and a function call stack is associated with the process executing a function, not with the code for the function.*

The important point is: an operating system must allocate additional storage for each process, even if the process shares the same code that other process(s) are using. As a consequence, the amount of memory available limits the number of processes that can be created.

## 2.7 Process Exit And Process Termination

The example in file *ex3.c* consists of a concurrent program with three processes: the initial process and the two processes that were started with the system call *create*. Recall that when it reached the end of the code in the main program, the initial process ceased execution. We use the term *process exit* to describe the situation. Each process begins execution at the start of a function. A process can exit by reaching the end of the function or by executing a return statement in the function in which it starts. Once a process exits, it disappears from the system; there is simply one less computation in progress.

Process exit should not be confused with normal function call and return or with recursive function calls. Like a sequential program, each process has its own stack of function calls. Whenever it executes a call, an activation record for the called function is pushed onto the stack. Whenever it returns, a function's activation record is popped

off the stack. Process exit occurs only when the process pops the last activation record (the one that corresponds to the top-level function in which the process started) off its stack.

The system routine *kill* provides a mechanism to *terminate* a process without waiting for the process to exit. In a sense, *kill* is the inverse of *create* — *kill* takes a process ID as an argument, and removes the specified process immediately. A process can be killed at any time and at any level of function nesting. When terminated, the process ceases execution and local variables that have been allocated to the process disappear; in fact, the entire stack of functions for the process is removed.

A process can exit by killing itself as easily as it can kill another process. To do so, the process uses system call *getpid* to obtain its own process ID, and then uses *kill* to request termination:

```
kill( getpid() );
```

When used to terminate the current process, the call to *kill* never returns because the calling process exits.

## 2.8 Shared Memory, Race Conditions, And Synchronization

In Xinu, each process has its own copy of local variables, function arguments, and function calls, but all processes share the set of global (external) variables. Sharing data is sometimes convenient, but it can be dangerous, especially for programmers who are unaccustomed to writing concurrent programs. For example, consider two concurrent processes that each increment a shared integer, *n*. In terms of the underlying hardware, incrementing an integer requires three steps:

- Load the value from variable *n* in memory into a register
- Increment the value in the register
- Store the value from the register back into the memory location for *n*

Because the operating system can choose to switch from one process to another at any time, a potential *race condition* exists in which two processes attempt to increment *n* at the same time. Process 1 might start first and load the value of *n* into a register. But just at that moment, the operating system switches to process 2, which loads *n*, increments the register, and stores the result. Unfortunately, when the operating system switches back to process 1, execution resumes with the original value of *n* in a register. Process 1 increments the original value of *n* and stores the result to memory, overwriting the value that process 2 placed in memory.

To see how sharing works, consider the code in file *ex4.c*. The file contains code for two processes that communicate through a shared integer, *n*†. One process repeatedly increments the value of the shared integer, while the other process repeatedly prints the value.

†The code uses the type name *int32* to emphasize that variable *n* is a 32-bit integer; a later section explains conventions for type names.

```
/* ex4.c - main, produce, consume */

#include <xinu.h>

void    produce(void), consume(void);

int32   n = 0;          /* external variables are shared by all processes */

/*------------------------------------------------------------------------
 *  main  --  example of unsynchronized producer and consumer processes
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(consume, 1024, 20, "cons", 0) );
        resume( create(produce, 1024, 20, "prod", 0) );
}

/*------------------------------------------------------------------------
 * produce  --  increment n 2000 times and exit
 *------------------------------------------------------------------------
 */
void    produce(void)
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ )
                n++;
}

/*------------------------------------------------------------------------
 * consume  --  print n 2000 times and exit
 *------------------------------------------------------------------------
 */
void    consume(void)
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ )
                printf("The value of n is %d \n", n);
}
```

In the code, global variable *n* is a shared integer, initialized to zero. The process executing *produce* iterates 2000 times, incrementing *n*; we call this process the *produc-*

*er*. The process executing *consume* also iterates 2000 times. It displays the value of *n* in decimal; we call this process the *consumer*.

Try running file *ex4.c* — its output may surprise you. Most programmers suspect that the consumer will print at least a few, perhaps all, of the values between 0 and 2000, but it does not. In a typical run, *n* has the value 0 for the first few lines; after that, its value becomes 2000.† Even though the two processes run concurrently, they do not require the same amount of CPU time per iteration. The consumer process must format and write a line of output, an operation that requires hundreds of machine instructions. Although formatting is expensive, it does not dominate the timing; output does. The consumer quickly fills the available output buffers, and must wait for the output device to send characters to the console before it can proceed. While the consumer waits, the producer runs. Because it executes only a few machine instructions per iteration, the producer runs through its entire loop and exits in the short time it takes the console device to send a line of characters. When the consumer resumes execution again, it finds that *n* has the value 2000.

Production and consumption of data by independent processes is common. The question arises: how can a programmer synchronize producer and consumer processes so the consumer receives every data value produced? Clearly, the producer must wait for the consumer to access the data item before generating another. Likewise, the consumer must wait for the producer to manufacture the next item. For the two processes to coordinate correctly, a synchronization mechanism must be designed carefully. The crucial constraint is:

> *In a concurrent programming system, no process should use the CPU while waiting for another process.*

A process that executes instructions while waiting for another is said to engage in *busy waiting*. To understand our prohibition on busy waiting, think of the implementation. If a process uses the CPU while waiting, the CPU cannot be executing other processes. At best, the computation will be delayed unnecessarily, and at worst, the waiting process will use all the available CPU time in a single-CPU processor and wind up waiting forever.

Many operating systems include coordination functions that applications can use to avoid busy waiting. Xinu provides a *semaphore* abstraction — the system supplies a set of system calls that allow applications to operate on semaphores and to create semaphores dynamically. A semaphore consists of an integer value that is initialized when the semaphore is created and a set of zero or more processes that are waiting on the semaphore. The system call *wait* decrements a semaphore and adds the calling process to the set of waiting processes if the result is negative. The system call *signal* performs the opposite action by incrementing the semaphore and allowing one of the waiting process to continue, if any are waiting. To synchronize, a producer and consumer need two semaphores: one on which the consumer waits and one on which the producer waits. In Xinu, semaphores are created dynamically with the system call *semcreate*, which takes

†The example assumes a 32-bit architecture in which each operation affects the entire 32-bit integer; when run on an 8-bit architecture, some bytes of *n* may be updated before others.

the desired initial count as an argument, and returns an integer identifier by which the semaphore is known.

Consider the example in file *ex5.c*. The main process creates two semaphores, *consumed* and *produced*, and passes them as arguments to the processes it creates. Because the semaphore named *produced* begins with a count of 1, *wait* will not block the first time it is called in *cons2*. So, the consumer is free to print the initial value of *n*. However, semaphore *consumed* begins with a count of 0, so the first call to *wait* in *prod2* blocks. In effect, the producer waits for semaphore *consumed* before incrementing *n* to guarantee that the consumer has printed it. When the example executes, the producer and consumer coordinate, and the consumer prints all values of *n* from 0 through 1999.

```
/* ex5.c - main, prod2, cons2 */

#include <xinu.h>

void    prod2(sid32, sid32), cons2(sid32, sid32);

int32   n = 0;                  /* n assigned an initial value of zero  */

/*------------------------------------------------------------------------
 *  main  --  producer and consumer processes synchronized with semaphores
 *------------------------------------------------------------------------
 */
void    main(void)
{
        sid32   produced, consumed;

        consumed = semcreate(0);
        produced = semcreate(1);
        resume( create(cons2, 1024, 20, "cons", 2, consumed, produced) );
        resume( create(prod2, 1024, 20, "prod", 2, consumed, produced) );
}

/*------------------------------------------------------------------------
 * prod2  --  increment n 2000 times, waiting for it to be consumed
 *------------------------------------------------------------------------
 */
void    prod2(
          sid32         consumed,
          sid32         produced
        )
{
        int32   i;
```

```
        for( i=1 ; i<=2000 ; i++ ) {
                wait(consumed);
                n++;
                signal(produced);
        }
}

/*------------------------------------------------------------------------
 * cons2  --  print n 2000 times, waiting for it to be produced
 *------------------------------------------------------------------------
 */
void    cons2(
          sid32         consumed,
          sid32         produced
        )
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ ) {
                wait(produced);
                printf("n is %d \n", n);
                signal(consumed);
        }
}
```

## 2.9 Semaphores And Mutual Exclusion

Semaphores provide another important purpose, *mutual exclusion*. Two or more processes engage in mutual exclusion when they cooperate so that only one of them obtains access to a shared resource at a given time. For example, suppose two executing processes each need to insert items into a shared linked list. If they access the list concurrently, pointers can be set incorrectly. Producer–consumer synchronization does not handle the problem because the two processes do not alternate accesses. Instead, a mechanism is needed that allows either process to access the list at any time, but guarantees mutual exclusion so that one process will wait until the other finishes.

To provide mutual exclusion for use of a resource such as a linked list, the processes create a single semaphore that has an initial count of 1. Before accessing the shared resource, a process calls *wait* on the semaphore, and calls *signal* after it has completed access. The calls to *wait* and *signal* can be placed at the beginning and end of the procedures designed to perform the update, or they can be placed around the lines of code that access the shared resource. We use the term *critical section* to refer to the code that cannot be executed by more than one process at a time.

For example, file *ex6.c* shows a function that adds an item to an array that is shared by multiple concurrent processes. The critical section consists of the single line:

```
shared[n++] = item;
```

which references the array and increments the count of items. Thus, the code for mutual exclusion only needs to surround one line of code. In the example, the critical section has been placed in a function, *additem*, which means the calls to *wait* and *signal* occur at the beginning and end of the function.

The code in *additem* calls *wait* on semaphore *mutex* before accessing the array, and calls *signal* on the semaphore when access is complete. In addition to the function, the file contains declarations for three global variables: an array, *ary*, an index for the array, *n*, and the ID of the semaphore used for mutual exclusion, *mutex*.

```
/* ex6.c - additem */

#include <xinu.h>

sid32   mutex;                  /* assume initialized with semcreate    */
int32   shared[100];            /* an array shared by many processes    */
int32   n = 0;                  /* count of items in the array          */

/*------------------------------------------------------------------------
 * additem  --  obtain exclusive access to array ary and add an item to it
 *------------------------------------------------------------------------
 */
void    additem(
          int32         item    /* item to add to array ary             */
        )
{
        wait(mutex);
        shared[n++] = item;
        signal(mutex);
}
```

The code assumes that global variable *mutex* will be assigned the ID of a semaphore before any calls to *additem* occur. That is, during initialization, the following statement was executed:

```
mutex = semcreate(1);
```

The code in file *ex6.c* provides a final illustration of the difference between the way one programs in sequential and concurrent environments. In a sequential program, a function often acts to isolate changes to a data structure. By localizing the code that

changes a data structure in one function, a programmer gains a sense of security — only a small amount of code needs to be checked for correctness because nothing else in the program will interfere with the data structure. In a concurrent execution environment, isolating the code into a single function is insufficient. A programmer must guarantee that execution is exclusive because interference can come from another process executing the same function at the same time.

## 2.10 Type Names Used In Xinu

Data declarations in the code above illustrate conventions used throughout the text. For example, semaphores are declared using the type name *sid32*. This section explains the reasoning for the choice.

Two important questions arise when programming in C. When is it appropriate to define a new type name? How should a type name be chosen? The questions require careful thought because types fill two conceptual roles.

- Size. A type defines the storage associated with a variable and the set of values that can be assigned to the variable.
- Use. A type defines the abstract meaning of a variable and helps a programmer know how a variable can be used.

*Size*. The sizes of variables are especially important in an embedded system because a programmer must design data structures that fit in the memory available. Furthermore, choosing a size that does not match the underlying hardware can result in unexpected processing overhead (e.g., arithmetic operations on large integers can require multiple hardware steps). Unfortunately, C does not specify the exact size of concrete types, such as *int*, *short*, and *long*. Instead, the size of items depends on the underlying computer architecture. For example, a *long* integer can occupy 32 bits on one computer and 64 bits on another computer. To guarantee sizes, a programmer can define and use a set of type names, such as *int32*, that specify data size.

*Use*. The classic purpose of a type arises from the need to define the purpose of a variable (i.e., to tell how the variable is used). For example, although semaphore IDs are integers, defining a type name such as *semaphore* makes it clear to anyone reading the code that a variable holds a semaphore ID and should only be used where a semaphore ID is appropriate (e.g., as an argument to a function that operates on a semaphore). Thus, although it consists of an integer, a variable of type *semaphore* should not be used as a temporary value when computing an arithmetic expression, nor should it be used to store a process ID or a device ID.

Include files further complicate type declarations in C. In principle, one would expect each include file to contain the type, constant, and variable declarations related to a single module. Thus, one would expect to find the type for a process identifier declared in the include file that defines items related to processes. In an operating system, however, many cross-references exist among modules. For example, we will see that the in-

clude file for semaphores references the process type, and the include file for processes references the semaphore type.

We have chosen an approach in which types accommodate the need to define size as well as the need to define use. Instead of using the C form of *char*, *short*, *int*, and *long*, the code uses the types that Figure 2.1 lists.

For types that correspond to operating system abstractions, each name combines a short mnemonic that identifies the purpose plus a numeric suffix that identifies the size. Thus, a type that defines a semaphore ID to be a 32-bit integer has been assigned the name *sid32*, and a type that defines a queue ID to be a 16-bit integer has been assigned the name *qid16*.

| Type | Meaning |
|---|---|
| byte | unsigned 8-bit value |
| bool8 | 8-bit value used as a Boolean |
| int16 | signed 16-bit integer |
| uint16 | unsigned 16-bit integer |
| int32 | signed 32-bit integer |
| uint32 | unsigned 32-bit integer |

**Figure 2.1** Basic type names for integers used throughout Xinu.

To permit cross-references of types among modules, a single include file, *kernel.h*, contains declarations for all type names, including the types listed in Figure 2.1. Thus, each source file must include *kernel.h* before referencing any type name. In particular, an include for *kernel.h* must precede the includes for other modules. For convenience, a single include file, *xinu.h*, includes all header files used in Xinu in the correct order.

## 2.11 Operating System Debugging With Kputc And Kprintf

The examples in this chapter use Xinu functions *putc* and *printf* to display output on the *CONSOLE*. Although such functions work well once an operating system has been completed and tested, they are not used during construction or debugging because they require many components of the operating system to functioning correctly. What do operating system designers use?

The answer lies in *polled I/O*. That is, a designer creates a special I/O function that does not need interrupts to be working. Following Unix tradition, we call the special function *kputc* (i.e., a version of *putc* suitable for use inside the operating system kernel). *Kputc* takes a character, *c*, as an argument and performs four steps:

- Disable interrupts
- Wait for the CONSOLE serial device to be idle
- Send character *c* to the serial device
- Restore interrupts to their previous status

Thus, when a programmer invokes *kputc*, all processing stops until the character has been displayed. Once the character has been displayed, processing resumes. The important idea is that the operating system itself does not need to be working because *kputc* manipulates the hardware device directly.

Once *kputc* is available, it is easy to create a function that can display formatted output. Again following Unix tradition, we call the function *kprintf*. Basically, *kprintf* operates exactly like *printf* except that instead of invoking *putc* to display each character, *kprintf* invokes *kputc*.†

Although it is not important to understand the exact details of how polled I/O operates, it is essential to use polled I/O when debugging:

> *Whenever they modify or extend the operating system, programmers should use* kprintf *to display messages rather than* printf.

## 2.12 Perspective

Concurrent processing is one of the most powerful abstractions in Computer Science. It makes programming easier, less error prone, and in many cases, yields higher overall performance than code that attempts to switch among tasks manually. The advantages are so significant that once concurrent execution was introduced, it rapidly became the primary choice for most programming.

## 2.13 Summary

An understanding of an operating system begins with the set of services the system provides to applications. Unlike a conventional, sequential programming environment, an operating system provides concurrent execution in which multiple processes proceed at the same time. In our example system, as in most systems, a process can be created or terminated at run-time. Multiple processes can each execute a separate function, or multiple processes can execute a single function. In a concurrent environment, storage for arguments, local variables, and a function call stack is associated with each process rather than with the code.

†Debugging operating system code is difficult because disabling interrupts can change the execution of a system (e.g., by preventing clock interrupts). Thus, a programmer must be extremely careful when using *kprintf*.

Processes use synchronization primitives, such as semaphores, to coordinate execution. Two primary forms of coordination are producer–consumer synchronization and mutual exclusion.

## EXERCISES

**2.1** What is an API, and how is an API defined?

**2.2** To what does *multiprogramming* refer?

**2.3** List the two basic categories of multiprogramming systems, and state the characteristics of each.

**2.4** What characteristics are generally associated with the terms *process*, *task*, and *thread*?

**2.5** How is a process ID used?

**2.6** How does calling function *X* differ from calling *create* to start a process executing function *X*?

**2.7** The program in file *ex3.c* uses three processes. Modify the code to achieve the same results using only two processes.

**2.8** Test the program in file *ex4.c* repeatedly. Does it always print the same number of zeroes? Does it ever print a value of *n* other than 0 or 2000?

**2.9** In Xinu, what is the difference in storage between global variables and local variables?

**2.10** Why do programmers avoid busy waiting?

**2.11** Suppose three processes attempt to use function *additem* in file *ex6.c* at the same time. Explain the series of steps that occur, and give the value of the semaphore during each step.

**2.12** Modify the producer–consumer code in file *ex5.c* to use a buffer of 15 slots, and have the producer and consumer synchronize in such a way that a producer can generate up to 15 values before blocking and a consumer can extract all values in the buffer before blocking. That is, arrange for the producer to write integers 1, 2, ... in successive locations of the buffer, wrapping around to the beginning after filling the last slot, and have the consumer extract values and print them on the console. How many semaphores are needed?

**2.13** In file *ex5.c*, the semaphore *produced* is created with a count of 1. Rewrite the code so *produced* is created with a count of 0 and the producer signals the semaphore once before starting the iteration. Does the change affect the output?

**2.14** Find the documentation for the serial port (or console device hardware) on a platform to which you have access. Describe how to construct a polled I/O function, *kputc()* that uses the device.

# Chapter Contents

# 3

# An Overview Of The Hardware and Run-Time Environment

*One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man.*

— Elbert Hubbard

## 3.1 Introduction

Because it deals with the details of devices, processors, and memory, an operating system cannot be designed without knowledge of the capabilities and features of the underlying hardware. The example system in this text runs on a small embedded hardware platform, the E2100L Linksys wireless router. We chose the Linksys router because it offers simplicity, a well-known instruction set, availability, and low cost. The system is small enough to allow readers to understand most of the hardware, and sufficiently complex to illustrate how an operating system works on a general-purpose system. Finally, the E2100L allows programmers to download and run code without requiring a sophisticated hardware lab and without replacing ROM chips.

The remainder of the chapter introduces the Linksys hardware, describing pertinent features of the processor, memory, and I/O devices. The chapter explains the architecture, memory address space, the run-time stack, the interrupt mechanism, and device addressing. Although the details refer to the E2100L, the basic concepts apply broadly to most computer systems.

## 3.2 Physical And Logical Organizations Of The E2100L

Physically, a Linksys router consists of a small self-contained box that uses a separate power cube. Because Linksys markets the E2100L as a consumer product, the router comes completely assembled. Furthermore, most of the major components are contained in a single VLSI chip that is known as a *System On a Chip* (*SoC*).

Although the circuit board contains pins for a serial interface, no external connection is supplied. Before the serial interface can be used, an inexpensive serial converter must be connected to pins on the board. Information about a serial connection and instructions describing how to attach a serial connector can be found on the web site:

http://www.xinu.cs.purdue.edu

Logically, the E2100L follows the same overall architecture as most general-purpose computer systems. The components on the SoC include a processor, co-processor, memory interface, and I/O device interfaces. The I/O devices of interest include wired and wireless network devices.

One wired network interface connects to a controller that acts as a hub with four RJ-45 sockets that can connect to local computers.† The other wired network interface, intended for an Internet connection, attaches to a single RJ-45 socket. An internal bus, known as a *system backplane*, provides a central interconnect that allows components to interact. Figure 3.1 illustrates the conceptual organization.

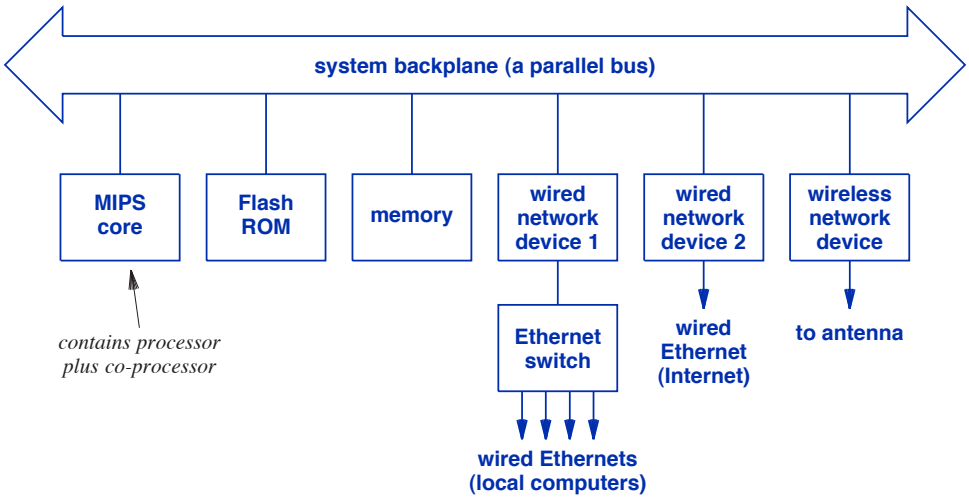



**Figure 3.1** The logical organization of major components in the E2100L.

†Hardware components on the chip can be reconfigured to provide alternative logical organizations. The chapter and the remainder of the text describe the default hardware configuration that is available when the device is powered on without giving details about alternatives.

The next sections consider features of the E2100L that are pertinent to an operating system. For now, we will focus on the overall design of each component and how the components fit together. Later chapters discuss additional details, explain how an operating system interacts with the hardware, and provide examples.

## 3.3 Processor Organization And Registers

Like many embedded systems, the E2100L uses a *RISC*† processor. The processor implements a *MIPS* instruction set. Except for a few special cases, an operating system does not need to focus on the instruction set because a compiler generates the necessary code.

The processor contains thirty-two *general-purpose registers*. Each register is 32 bits wide, and can hold an integer, an address, or four 8-bit characters. As in most RISC processors, many MIPS operations take arguments in registers and leave the result of the operation in a register. Despite being classified as general-purpose, individual registers are assigned specific uses by the compiler. For example, one of the registers is used as a stack pointer that is changed to allocate space for an activation record whenever a function call occurs. Figure 3.2 lists the registers by giving the name used to reference a register and the meaning assigned.

| Name | Meaning |
|---|---|
| 0 | Always contains zero |
| AT | Assembler temporary (reserved for assembler to use) |
| V0 and V1 | Return values from a function call |
| A0 – A3 | Argument registers for first 4 arguments |
| T0 – T9 | Temporary (i.e., not preserved) across a function call |
| S0 – S9 | Saved (i.e., preserved) across a function call |
| K0 and K1 | Kernel registers used by interrupt hardware |
| SP | Stack pointer (stack grows downward) |
| RA | Return address during a function call |

**Figure 3.2** The general-purpose registers and the meaning of each.

†RISC stands for Reduced Instruction Set Computer.

A co-processor contains a set of special-purpose registers that support many extra features. For example, because a RISC processor cannot perform division in one cycle, the co-processor handles 64-bit division. Consequently, the co-processor has a pair of registers that each store half of a 64-bit value. One register stores the high 32 bits and the other stores the low 32 bits. Similarly, the co-processor has registers that store: the current *interrupt mask* (telling whether interrupts are enabled), a return address for interrupts or exceptions, and debugging information. Chapter 12 on interrupt processing will show how an operating system uses specific co-processor registers.

## 3.4 Bus Operation: The Fetch-Store Paradigm

The bus, known as a *system backplane*, provides the primary path between the processor and other components, namely the memory, I/O devices, and other interface controllers. The system backplane uses a *fetch-store paradigm* typical of most computer system buses. For example, when it needs to access memory, the processor places an address on the bus and issues a *fetch* request to obtain the corresponding value. The memory hardware responds to the request by looking up the address in memory, placing the data value on the bus, and signalling the processor that the value is ready. Similarly, to store a value in memory, the processor places an address and value on the bus and issues a *store* request; the memory hardware extracts the value and stores a copy in the specified memory location. Bus hardware handles many details of the fetch-store paradigm, including signals that the processor and other components use to communicate and control access to the bus. We will see that an operating system can use a bus without knowing many details of the underlying hardware.

The system uses *memory-mapped I/O*, which means that each I/O device is assigned a set of addresses in the bus address space, and the processor uses the same fetch-store paradigm to communicate with I/O devices as with memory. As we will see, communication with a memory-mapped I/O device resembles data access. First, the processor computes the address associated with a device. Second, to access the device, the processor either stores a value to the address or loads a value from the address into a register.

## 3.5 Direct Memory Access

Some I/O devices on the E2100L offer *Direct Memory Access* (*DMA*), which means the device contains hardware that can use the bus to communicate directly with memory. The key idea is that DMA allows I/O to proceed quickly because it does not interrupt the CPU frequently nor does it require the processor to control each data transfer. Instead, a processor can give the I/O device a list of operations, and the device proceeds from one to the next. Thus, DMA allows a processor to continue running code while the device operates.

As an example, consider how a network device uses DMA. To receive a packet, the operating system allocates a buffer in memory and starts the network device. When a packet arrives, the device hardware transfers the packet directly into the buffer in memory and interrupts the processor. To send a packet, the operating system places the packet in a buffer in memory and starts the device. The device fetches the packet from the buffer in memory and transmits the packet on the network.

In addition to single packet transfers, DMA hardware on the E2100L allows the processor to request multiple operations. In essence, the processor creates a list of packets to be sent and a list of buffers to be used for incoming packets. The network interface hardware uses the lists to send and receive packets without requiring the processor to restart the device after each operation. As long as the processor consumes incoming packets faster than they arrive and adds fresh buffers to the list, the network hardware device will continue to read packets. Similarly, as long as the processor continues to generate packets and add them to the list, the network hardware device will continue to transmit them. Later chapters explain more DMA details, and the example code illustrates how a device driver in the operating system allocates I/O buffers and controls DMA operations.

## 3.6 The Bus Address Space

The system backplane uses a 32-bit bus address space, with addresses ranging from 0x00000000 through 0xFFFFFFFF. Some of the addresses in the address space correspond to memory, some to FlashROM, and others to I/O devices. To accommodate operating systems that support memory protection and virtual memory, the address space is divided in half, with the lower half (addresses 0x00000000 0x7FFFFFFF) comprising a *user space* and the upper half (addresses 0x80000000 through 0xFFFFFFFF) comprising the *kernel space*, which is further divided into *segments*. Figure 3.3 illustrates the organization.

Memory on the E2100L is divided into 8-bit *bytes*, with a byte being the smallest addressable unit. The C language uses the term *character* in place of *byte* because each byte can hold one ASCII character. Although a 32-bit bus can address 4 Gbytes, the E2100L does not contain 4 Gbytes of memory. Instead, the physical memory occupies 16 Mbytes that appear to be repeated in the address space. The next section explains the replication.

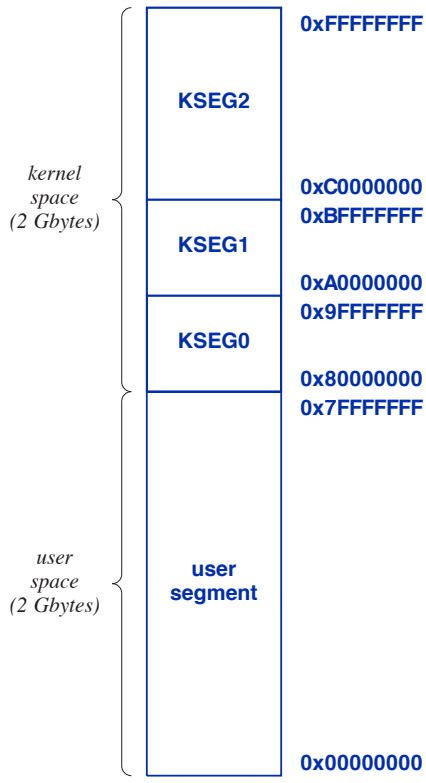



**Figure 3.3** The division of the bus address space into user and kernel areas, with further division into segments.

## 3.7 Contents Of Kernel Segments KSEG0 and KSEG1

Segments KSEG0 and KSEG1 are fundamental to the operating system, and have special meaning. The operating system resides in KSEG0. The highest addresses of the segment are reserved for I/O devices; lower addresses correspond to physical memory. Figure 3.4 illustrates the layout of KSEG0.

Although KSEG0 contains addresses for 512 Mbytes, the physical memory on the E2100L comprises only 16 Mbytes. The physical memory appears to be replicated through the address space. For example, physical memory occupies relative addresses 0x0000000 through 0x000FFFFF. The same physical memory appears again in locations 0x00100000 through 0x001FFFFF. That is, the hardware ignores high-order bits of an address when mapping the address to memory.†

†Although embedded systems tend to ignore high-order address bits, many computer systems restrict addresses to physical memory, and classify other references as errors.

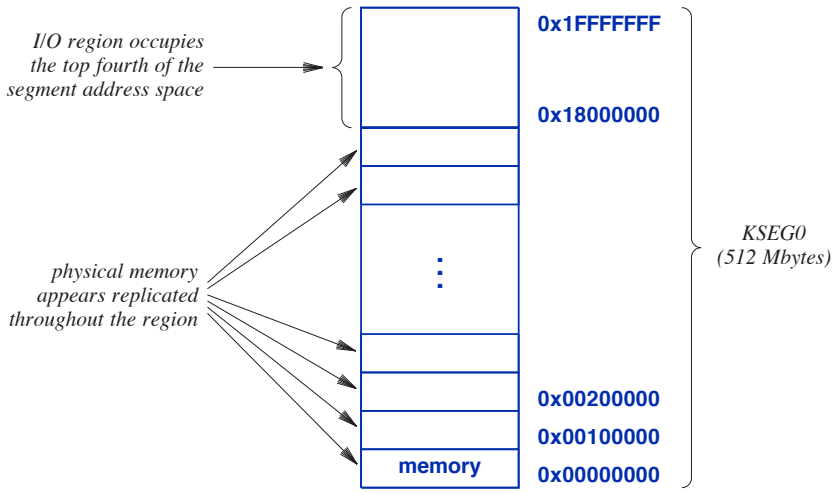



**Figure 3.4** The KSEG0 address space.

The important point is:

> *Referencing an address beyond physical memory may not cause an error to occur. Instead, high-order bits of the address are ignored, and the address may map into a valid reference as if the physical memory is replicated.*

## 3.8 Bus Startup Static Configuration

Desktop computers contain complex bus hardware that allows a processor to discover components that attach to the bus. Each component is assigned a unique identifier that specifies the vendor and the specific type. At startup, an operating system probes the bus to determine which components are present, and each component returns its ID. The mechanism makes it possible for an operating system to configure itself dynamically, allowing a copy of the system to run on a variety of hardware platforms.

Unlike desktop systems, embedded systems often use *static configuration*. That is, the complete hardware configuration is known when an operating system is designed, and the system does not discover hardware at startup, nor does it reconfigure. Chapter 24 discusses system configuration in more detail, and shows an example of static configuration.

## 3.9 Calling Conventions And The Run-Time Stack

Function invocation forms a key aspect of operating systems. Application programs use function calls to invoke operating system services such as creating concurrent processes and performing I/O. As it switches context from one process to another, the operating system must manage separate sets of function calls. The following paragraphs define key concepts related to function invocation.

*Calling conventions*. The steps taken during a function call and return are known as *calling conventions*. The term *convention* arises because the hardware does not dictate all details. Instead, the hardware design places some constraints on possible approaches, and leaves many choices to a compiler writer. We will see that because it calls functions to process interrupts and switches from one running process to another, the operating system must understand and follow the same conventions as the compiler.

*Run-time stack*. A statically scoped language, such as C, uses a *run-time stack* to store the state associated with a function call. The compiler allocates enough space on the stack to hold an *activation record* for the called function. The allocated space is known as a *stack frame*. Each activation record contains space for the local variables associated with the function, temporary storage needed during computation, a return address, and other miscellaneous items. Also by convention, a stack grows downward from higher memory addresses to lower memory addresses. Thus, when a function call occurs, the stack grows downward in memory to hold the activation record for the function call.

*Arguments*. When a function is called, the caller supplies a set of actual *arguments* that correspond to *parameters*. Most RISC architectures arrange a fixed number of arguments to be passed in registers and the remaining arguments to be passed in memory. The example code uses registers A0 through A3 to pass the first four arguments; arguments beyond the first four are placed in the activation record on the stack.

*Stack frame contents*. A compiler computes the size needed for a stack frame, and handles assigning space to each local variable. However, an operating system needs to know exactly how arguments are stored. In the example code in the text, the highest word in each stack frame is used to store a return address, the lowest four words are reserved as an argument save area, and successive words beyond the argument save area contain arguments beyond the first four arguments. Figure 3.5 illustrates the format.

Interestingly, the argument save area that occupies the lowest four words of a stack frame for function, *f*, is not used by *f*. Instead, the area is reserved for use by functions that *f* calls. In the figure, if function *X* calls function *Y*, the lowest four words of the stack for function *X* will be used by function *Y*.

To understand why an argument save area is needed, observe that the first four arguments are passed in registers A0 through A3. For example, suppose function *X* calls function *Y*, and passes an argument *q*. Register A0 will contain the value of *q* when *Y* begins execution. If function *Y* calls function *Z* and passes argument *r*, register A0 will be used. That is, to pass argument *r* to function *Z*, *Y* must overwrite the value *q* that it received from *X*. To insure that *q* is not lost, *Y* saves the value of *2* on the stack before calling *Z*, and then restores the value after the call returns. In theory, *Y* could save A0

at any location in memory — saving arguments in the caller's stack frame is merely a convention adopted by the compiler being used.

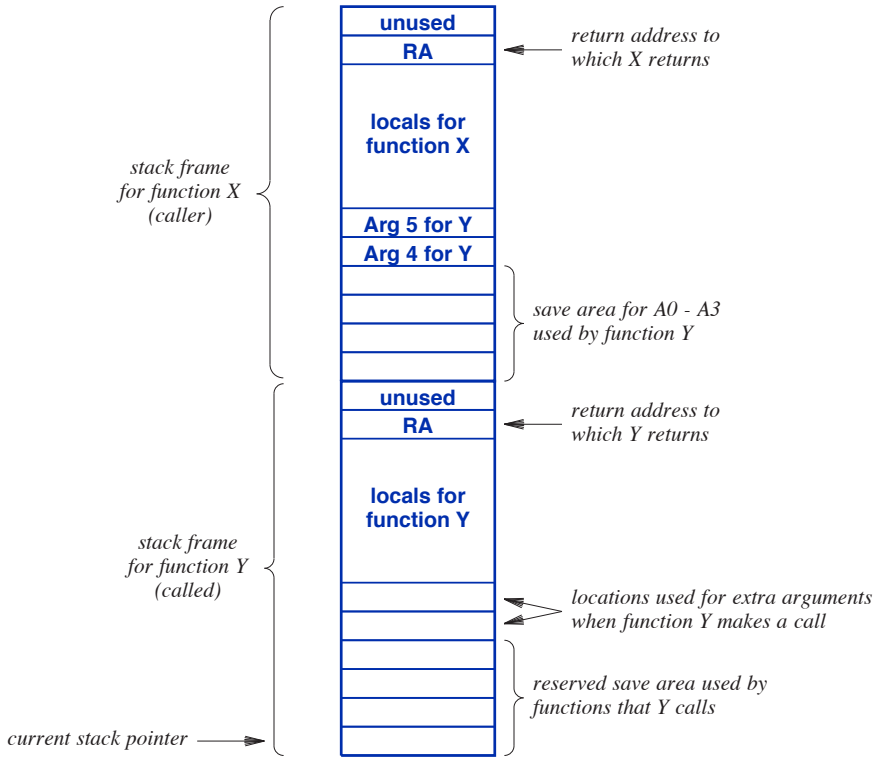



**Figure 3.5** Format of two stack frames when function X calls function Y.

As a further convention, the *frame pointer* register is not used. Instead, the compiler computes the exact size of the stack frame that will be needed for each function (including the four word argument save area), and arranges code to decrement the stack pointer by the appropriate size when a function is called. Because a RISC processor does not include instructions that push values onto the stack, the compiler can compute the address of each local variable as a fixed offset beyond the stack pointer.

To summarize:

> *When referencing local variables or arguments on the run-time stack, the operating system follows the calling conventions used by the compiler. In the example code throughout the text, all references are given as offsets beyond the stack pointer register.*

## 3.10 Interrupts And Interrupt Processing

Modern processors provide mechanisms that allow external I/O devices to *interrupt* computation. Often, a processor has a related *exception* mechanism that is used to inform the software when an error or fault occurs (e.g., an application attempts division by zero or references a page in virtual memory that is not present in memory). From an operating system's point of view, interrupts are fundamental because they allow the CPU to perform computation at the same time I/O proceeds.†

Any of the I/O devices connected to a bus can interrupt the processor when the device needs service. To do so, the device places a signal on one of the bus control lines. During normal execution of the fetch-execute cycle, hardware in the processor monitors the control line and initiates interrupt processing when the control line has been signaled. In the case of a RISC processor, the main processor does not contain the hardware to handle interrupts. Instead, a co-processor performs much of the basic bus interaction and interrupt processing on behalf of the main processor.

For example, when a device interrupts, a MIPS processor performs three key steps:

- Sets a control bit that disables further interrupts
- Records the address of the instruction that is about to execute
- Jumps to the reserved location 0x80000180

The first step insures that while it is processing an interrupt from one device, the operating system will not be interrupted by another device. The second step provides a way for an operating system to return to normal execution once the interrupt has been processed. The third step allows the operating system to gain control whenever an interrupt occurs. Before an interrupt occurs, the operating system must store interrupt processing code at the reserved location (0x80000180). The compiler and loader start the operating system at location 0x80010000 (i.e., above the interrupt location), which guarantees that code can be stored in the reserved location without affecting other values in the operating system. At system startup, our example system stores a few instructions starting at the reserved location, which causes the processor to jump to an operating system interrupt function whenever an interrupt occurs.

While it manipulates global data structures and I/O queues, an operating system must prevent interrupts from occurring. The hardware supplies mechanisms to control interrupts. For example, one of the registers in the co-processor is an *interrupt mask* that the operating system uses to specify which devices are permitted to interrupt. Each bit in the mask corresponds to an interrupt source in the system; a source can consist of devices on the system bus, internal timers, or can arise from the execution of special processor opcodes. The initial value of all bits is zero, which means the corresponding source cannot interrupt. When it starts an I/O device, the operating system sets the corresponding mask bit to one, which allows the device to interrupt.

In addition to an individual bit for each device, the interrupt mask contains a *global interrupt status* bit. Setting the global bit to zero prevents all interrupts, despite the

†Later chapters explain how an operating system manages interrupt processing, and show how the high-level I/O operations a user performs relate to low-level device hardware mechanisms.

individual device bits. We will see that functions *disable* and *restore* manipulate the global status bit to allow the operating system to prevent interrupts temporarily and later re-enable interrupts.

Figure 3.6 lists the co-processor registers associated with interrupt processing and describes the purpose of each.

| Register | Purpose |
|---|---|
| STATUS | Interrupt status, including an EXL bit that specifies whether an interrupt is currently being processed, a global bit that tells whether all interrupts are disabled, and one interrupt enable bit for each source |
| CAUSE | Bits that identify the cause of an exception or the the source of an interrupt |
| EPC | Exception Program Counter that specifies the address at which normal processing should resume after an interrupt has been handled |

**Figure 3.6** Hardware registers in the co-processor associated with interrupt processing.

The processor interrogates co-processor control registers to determine which device caused the interrupt, and then interacts with the device. Once the interrupt has been handled, the processor can execute an *interrupt return* instruction to resume normal processing. Later chapters provide examples of interrupt processing and the steps taken to interact with various devices.

## 3.11 Exception Processing

Although exceptions are generated by the processor rather than by a separate I/O device, the MIPS hardware integrates exception handling with interrupt handling. That is, an exception causes the hardware to take the same steps as an interrupt: setting the EXL bit to disable further interrupts, recording the address of the instruction that caused the exception in the *EPC* register, and jumping to location 0x80000180.

One minor difference occurs between the way the processor handles interrupts and exceptions. When an interrupt occurs, the processor has completed executing one instruction, and is about to execute another. Thus, the hardware uses the *EPC* register to record the address of the next instruction to be executed. When an exception occurs, however, an instruction is currently being executed when the exception arises. Thus, the hardware uses the *EPC* register to record the address of the instruction being execut-

ed. When the processor returns from the exception, the instruction will be restarted. For example, if a page fault occurs, the exception handler reads the missing page from memory, and returns to the same point at which the exception occurred to restart the instruction that caused the fault.

## 3.12 Timer Hardware

In addition to external I/O devices, the E2100L hardware includes a *timer device*. When it expires, the timer generates an interrupt, which means that once it has enabled timer interrupts, the processor must be prepared to handle an interrupt.

On some embedded systems, all timer functions are implemented using a real-time hardware clock that generates interrupts regularly (e.g., 60 times per second). On the E2100L, however, the timer consists of two registers that the processor can load:

- Counter: the counter register is set to an initial value
- Limit: the limit register specifies the length of time to wait

The hardware uses the CPU clock, and increments the counter register once per cycle. When the counter reaches the limit, the hardware generates a timer interrupt.

Both the real-time clock approach and the E2100L's timer mechanism have advantages. The chief advantage of the timer mechanism arises from fewer interrupts — unlike a real-time clock, which interrupts continually, a timer only interrupts when a preset timeout has occurred. The real-time clock approach has the advantage of being able to relate interrupts directly to actual time instead of processor clock cycles. Of course, if one knows the processor clock rate, it is possible to convert to real time. Unfortunately, the calculation depends on the CPU speed, which means the operating system cannot be ported to a faster processor without changing the constant used in the conversion.

## 3.13 Serial Communication

Serial communication devices are among the simplest I/O devices available, and have been used on computers for decades. The E2100L contains an RS-232 serial communication device that is used as a system console. As with most serial devices, the serial hardware on the E2100L handles both input and output (i.e., both transmission and reception of characters). When an interrupt occurs, the processor must examine a device register to determine whether the output side has completed transmission or the input side has received a character. Chapter 15 examines serial interrupts in detail.

## 3.14 Polled vs. Interrupt-Driven I/O

Most I/O performed by an operating system uses the interrupt mechanism. The operating system interacts with the device to start an operation (either input or output), and then proceeds with computation. When the I/O operation completes, the device interrupts the processor, and the operating system can choose to start another operation.

Although they optimize concurrency and permit multiple devices to proceed in parallel with computation, interrupts cannot always be used. For example, consider displaying a startup message for a user before the operating system has initialized interrupts and I/O. Also consider a programmer who needs to debug new I/O code. In either case, interrupts cannot be used.

The alternative to *interrupt-driven I/O* is known as *polled I/O*. When using polled I/O, the processor starts an I/O operation, but does not enable interrupts. Instead, the processor enters a loop that repeatedly checks a device status register to determine whether the operation has completed. We have already seen an example of how an operating system designer can use polled I/O when we examined functions *kputc* and *kprintf* in Chapter 2.

## 3.15 Memory Cache And KSEG1

Recall that although KSEG1 follows KSEG0 in the address space, memory locations appear to be replicated. Thus, the first byte of KSEG1 points to the same physical memory as the first byte of KSEG0.

Despite the appearance of replication, the hardware enforces an important distinction between the two memory segments:

> *When the processor references an address in KSEG0, the reference goes to the L1 memory cache before being passed to the bus (i.e., system backplane); when the processor references an address in KSEG1, the reference is passed directly to the bus.*

For normal data references, a memory cache provides an important optimization — if the processor makes multiple references to an address within a short time span, the cache hardware returns the value faster than a memory reference. When performing I/O, however, a cache produces incorrect results. For example, consider the code that checks the status of a device when using a polled I/O paradigm. If a cache remembers and returns the previous value without accessing the device, the processor will not receive an accurate report of the device status. Thus, an operating system adheres to the following straightforward rule:

*To avoid obtaining stale values from the memory cache, each I/O reference, including addresses specified for DMA, must use KSEG1.*

## 3.16 Storage Layout

When it compiles a program, a C compiler partitions the resulting image into four memory segments:

- Text segment
- Data segment
- Bss segment
- Stack segment

The *text segment*, which includes code for the main program and all functions, occupies the lowest part of the address space. The *data segment*, which contains all initialized data, occupies the next region of the address space. The uninitialized data segment, called the *bss segment*, follows the data segment. Finally, the *stack segment* occupies the highest part of the address space and grows downward. Figure 3.7 illustrates the conceptual organization.

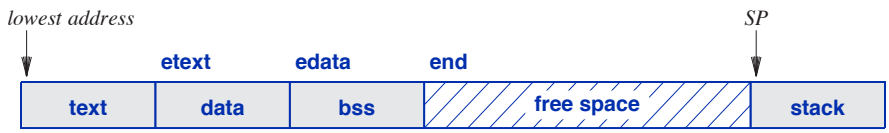



**Figure 3.7** Illustration of memory segments created by a C compiler.

The symbols *etext*, *edata*, and *end* in the figure refer to global variables that the loader inserts into the object program. The names are initialized to the first address beyond the text, data, and bss segments, respectively. Thus, a running program can determine how much memory remains between the end of the bss segment and the current top of stack by subtracting the address of *end* from the stack pointer, SP.

Chapter 9 explains memory allocation for multiple processes. Although all processes share the text, data, and bss segments, a separate stack segment must be allocated for each process. If three processes are executing, the stacks are allocated contiguously from the highest memory address downward as Figure 3.8 illustrates.

As the figure indicates, each process has its own stack pointer. At a given time, the stack pointer for process *i* must point to an address within the stack that has been allocated for process *i*. Later chapters explain the concept in detail.


**Figure 3.8** Illustration of memory with stack segments for three processes.

## 3.17 Memory Protection

The memory hardware available on the E2100L has multiple segments that can be used to provide protection for an operating system. Applications can be configured to run in *user mode*, which means they cannot read or write kernel memory. When an application makes a system call, control transfers to the kernel and the privilege level is increased to *kernel mode* until the call returns. The key to understanding protection is to remember that control can only transfer to the operating system at the specific entry points that the operating system designer provides. Thus, a designer can insure that an application only receives carefully controlled services.

Like most embedded systems, our example system avoids the complexity and run-time overhead of memory protection. Instead the code runs completely in KSEG0, and each process runs with kernel privilege. The lack of protection means a programmer must be careful because any process can access any memory location, including the memory allocated to operating system structures or memory allocated to another process's stack. If a process overflows the allocated stack area, the process's run-time stack will overwrite data in another process's stack. A later chapter discusses one technique software can use to help detect overflow.

## 3.18 Perspective

The hardware specifications for a processor or I/O device contain so many details that studying them can seem overwhelming. Fortunately, many of the differences among processors are superficial — fundamental concepts apply across most hardware platforms. Therefore, when learning about hardware, it is important to focus on the overall architecture and design principles rather than on tiny details.

## EXERCISES

**3.1** Some systems use a *programmable* interrupt address mechanism that allows the system to choose the address to which the processor jumps when an interrupt occurs. What is the advantage of a programmable interrupt address?

**3.2** DMA introduces the possibility of unexpected errors. What happens if a DMA operation that transfers *N* bytes of data begins at a memory location less than *N* bytes from the highest memory address?

**3.3** Read about hardware that uses multi-level interrupts. Should an interrupt at one level be able to interrupt the system while it is processing an interrupt from another level? Explain.

**3.4** What are the advantages of the memory layout shown in Figure 3.7? Are there disadvantages? What other layouts might be useful?

**3.5** Embedded hardware often includes multiple independent timers, each with its own interrupt source. Why might multiple timers be helpful? Can a system with only a single timer accomplish the same tasks as a system with multiple timers? Explain.

**3.6** If you are familiar with an assembly language, read about the calling conventions that are used to permit recursive function calls. Build a function that makes recursive calls, and demonstrate that your function works correctly.

# NOTES

# Chapter Contents

# 4

# List and Queue Manipulation

*As some day it may happen that a victim must be found, I've got a little list ....*

— W. S. Gilbert

## 4.1 Introduction

Linked list processing is fundamental in operating systems, and pervades each component. Linked structures enable a system to manage sets of objects efficiently without searching or copying. As we will see, process management is especially important.

This chapter introduces a set of functions that form the backbone of a linked list manipulation system. The functions represent a unified approach — a single data structure and a single set of nodes used by all levels of the operating system to maintain lists of processes. We will see that the data structure includes functions to create a new list, insert an item at the tail of a list, insert an item in an ordered list, remove the item at the head of a list, or remove an item from the middle of a list.†

The linked list functions are easy to understand because they assume that only one process executes a list function at a given time. Thus, a reader can think of the code as being part of a sequential program — there is no need to worry about interference from other processes executing concurrently. In addition, the example code introduces several programming conventions used throughout the text.

†Although linked list manipulation is usually covered in texts on data structures, the topic is included here because the data structure is unusual and because it forms a key part of the system.

## 4.2 A Unified Structure For Linked Lists Of Processes

A process manager handles objects called *processes*. Although at any time a process appears on only one list, a process manager moves a process from one list to another frequently. In fact, a process manager does not store all details about a process on a list. Instead, the process manager merely stores a process ID, a small, nonnegative integer used to reference the process. Because it is convenient to think of placing a process on a list, we will use the terms *process* and *process ID* interchangeably throughout the chapter.

An early version of Xinu had many lists of processes, each with its own data structure. Some consisted of first-in-first-out (FIFO) queues, and others were ordered by a key. Some lists were singly linked; others needed to be doubly linked to permit items to be inserted and deleted at arbitrary positions efficiently. After the requirements had been formulated, it became clear that centralizing the linked-list processing into a single data structure would reduce code size and eliminate many special cases. That is, instead of six separate sets of linked list manipulation functions, a single set of functions was created to handle all situations.

To accommodate all cases, a representation was selected with the following properties:

- All lists are doubly linked, which means a node points to its predecessor and successor.
- Each node stores a key as well as a process ID, even though a key is not used in a FIFO list.
- Each list has head and tail nodes; the head and tail nodes have the same memory layout as other nodes.
- Non-FIFO lists are ordered in descending order; the key in a head node is greater than the maximum valid key value, and the key value in the tail node is less than the minimum valid key.

Figure 4.1 illustrates the conceptual organization of a linked list data structure by showing an example list with two items.

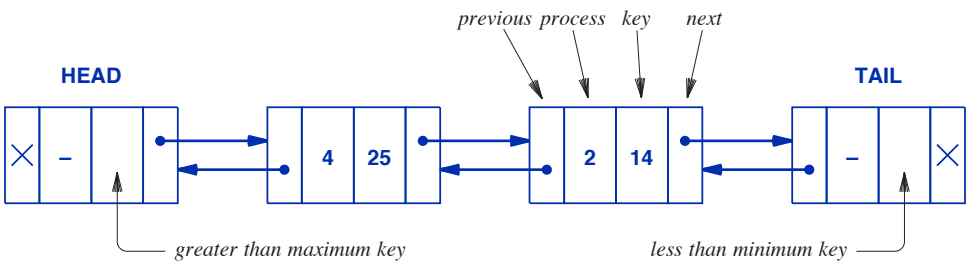



**Figure 4.1** The conceptual organization of a doubly-linked list containing processes 4 and 2 with keys 25 and 14, respectively.

As expected, the successor of the tail and the predecessor of the head are null. When a list is empty, the successor of the head is the tail and the predecessor of the tail is the head, as Figure 4.2 illustrates.

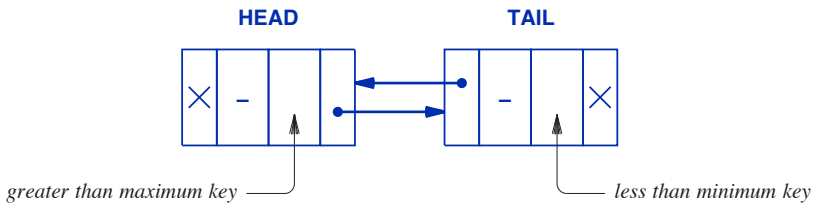



**Figure 4.2** The conceptual form of an empty linked list.

## 4.3 A Compact List Data Structure

One of the key design goals in an embedded system involves reducing the memory used. Instead of using a conventional implementation of a linked list, Xinu optimizes the memory required in two ways:

- Relative pointers
- Implicit data structure

To understand the optimizations, it is important to know that most operating systems place a fixed upper bound on the number of processes in the system. In Xinu, constant *NPROC* specifies the number of processes, and process identifiers range from *0* through *NPROC – 1*. In most embedded systems, *NPROC* is small (less than 50); we will see that a small limit makes each optimization work well.

*Relative pointers*. To understand the motivation for relative pointers, consider the space a conventional pointer occupies. On a 32-bit architecture, each pointer occupies four bytes. If the system contains fewer than 50 nodes, however, the size required can be reduced by placing nodes in contiguous memory locations and using a value between 0 and 49 as a reference. That is, the nodes can be allocated in an array, and the array index can be used instead of a pointer.

*Implicit data structure*. The second optimization focuses on omitting the process ID field from all nodes. Such an omission is feasible because:

A process appears on at most one list at any time.

To omit the process ID, use an array implementation and use the $i^{th}$ element of the array for process ID $i$. Thus, to put process 3 on a particular linked list, insert node 3 onto the list. Thus, the relative address of a node is the same as the ID of the process being stored.

Figure 4.3 illustrates how the linked list in Figure 4.1 can be represented in an array that incorporates relative pointers and implicit identifiers. Each entry in an array has three fields: a key, the index of the previous node, and the index of the next node. The head of the list has index 60, and the tail has index 61.

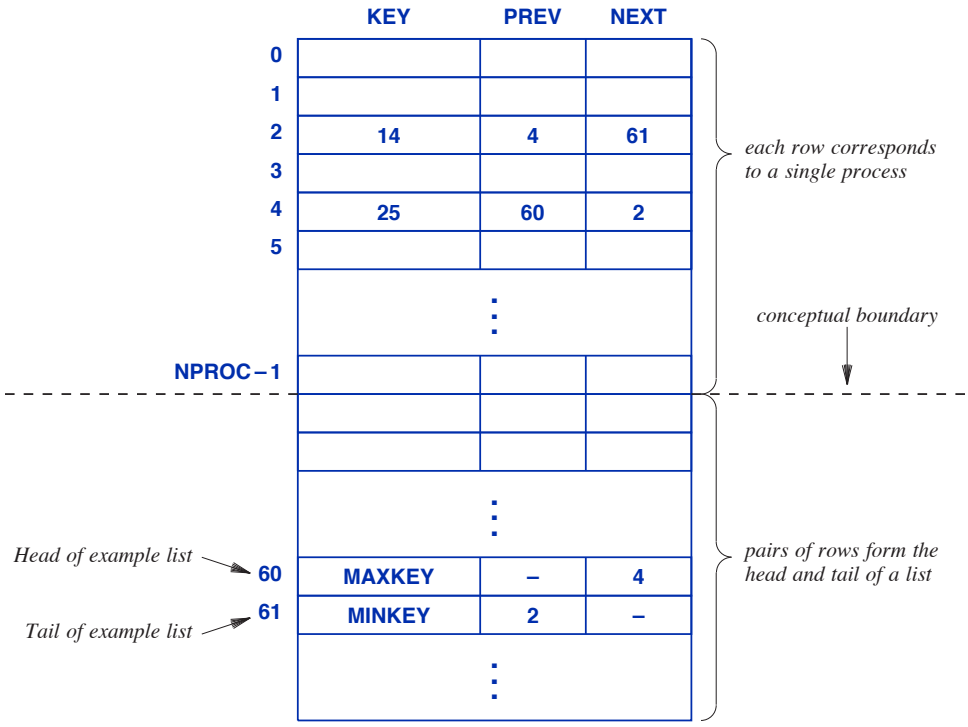



**Figure 4.3** The list from Figure 4.1 stored in the queue table array.

Because a *NEXT* or *PREV* field contains a relative pointer (i.e., an array index), the size of the field depends on the size of the array. For example, if the array contains fewer than 256 items, a single byte can be used to store an index.

Xinu uses the term *queue table* to refer to the array. The key to understanding the structure is to observe that array elements with an index less than *NPROC* differ from elements with a higher index. Positions 0 through *NPROC–1* each correspond to one process in the system; positions *NPROC* and higher are used to hold head and tail pointers for lists. Such a data structure is only feasible because the maximum number of processes and the maximum number of lists are each known at compile time and a process can only appear on one list at a given time.

## 4.4 Implementation Of The Queue Data Structure

To place process *i* on a list, the node with index *i* is linked into the list. A closer look at the code will make the operations clear. In Xinu, the queue table pictured above is named *queuetab*, and declared to be an array of *qentry* structures. File queue.h contains the declarations of both *queuetab* and *qentry*:

```
/* queue.h - firstid, firstkey, isempty, lastkey, nonempty              */

/* Queue structure declarations, constants, and inline functions        */

/* Default # of queue entries: 1 per process plus 2 for ready list plus */
/*                      2 for sleep list plus 2 per semaphore           */
#ifndef NQENT
#define NQENT   (NPROC + 4 + NSEM + NSEM)
#endif

#define EMPTY   (-1)            /* null value for qnext or qprev index  */
#define MAXKEY  0x7FFFFFFF      /* max key that can be stored in queue  */
#define MINKEY  0x80000000      /* min key that can be stored in queue  */

struct  qentry  {               /* one per process plus two per list    */
        int32   qkey;           /* key on which the queue is ordered    */
        qid16   qnext;          /* index of next process or tail        */
        qid16   qprev;          /* index of previous process or head    */
};

extern  struct qentry   queuetab[];

/* Inline queue manipulation functions */

#define queuehead(q)    (q)
#define queuetail(q)    ((q) + 1)
#define firstid(q)      (queuetab[queuehead(q)].qnext)
#define lastid(q)       (queuetab[queuetail(q)].qprev)
#define isempty(q)      (firstid(q) >= NPROC)
#define nonempty(q)     (firstid(q) <  NPROC)
#define firstkey(q)     (queuetab[firstid(q)].qkey)
#define lastkey(q)      (queuetab[ lastid(q)].qkey)

/* Inline to check queue id assumes interrupts are disabled */

#define isbadqid(x)     (((int32)(x) < 0) || (int32)(x) >= NQENT-1)
```

```
/* Queue function prototypes */

pid32   getfirst(qid16);
pid32   getlast(qid16);
pid32   getitem(pid32);
pid32   enqueue(pid32, qid16);
pid32   dequeue(qid16);
status  insert(pid32, qid16, int);
status  insertd(pid32, qid16, int);
qid16   newqueue(void);
```

The *queuetab* array contains *NQENT* entries. As Figure 4.3 indicates, an important implicit boundary occurs between element *NPROC–1* and element *NPROC*. Each element below the boundary corresponds to a process ID, and elements *queuetab[NPROC]* through *queuetab[NQENT]* correspond to the heads or tails of lists.

File *queue.h* introduces several features of C and conventions used throughout the book. Because the name ends in *.h*, the file will be included in other programs ("h" stands for *header*). Such files often contain the declarations for global data structures, symbolic constants, and inline functions (macros) that operate on the data structures. File *queue.h* defines *queuetab* to be an external variable (i.e., global), which means that every process will be able to access the array. The file also defines symbolic constants used with the data structure, such as constant *EMPTY* that is used to define an empty list.

Symbolic constant *NQENT*, which defines the total number of entries in the *queuetab* array, provides an example of conditional definition. The statement *#ifndef NQENT* means "compile the code down to the corresponding *#endif*, if and only if *NQENT* has not been defined previously." Thus, the code in *queue.h* assigns a value to *NQENT* only if it has not been defined previously. The value assigned,

$$NPROC + 4 + NSEM + NSEM$$

allocates enough entries in *queuetab* for *NPROC* processes plus head and tail pointers for *NSEM* semaphore lists, a ready list, and a sleep list. Conditional compilation is used to permit the size of the *queuetab* array to be changed without modifying the *.h* file.

The contents of entries in the *queuetab* array are defined by structure *qentry*. The file contains only a declaration of the elements in the *queuetab* array; Chapter 22 explains how data structures are initialized at system startup. Field *qnext* gives the relative address of the next node on a list, field *qprev* points to the previous node, and field *qkey* contains an integer key for the node. When a field, such as a forward or backward pointer, does not contain a valid index value, the field is assigned the value *EMPTY*.

## 4.5 Inline Queue Manipulation Functions

The functions *isempty* and *nonempty* are predicates (Boolean functions) that test whether a list is empty or not empty, given the index of its head as an argument. *Isempty* determines whether a list is empty by checking to see if the first node on the list is a process or the list tail; *nonempty* makes the opposite test. Remember that an item is a process if and only if its index is less than *NPROC*.

The other inline functions should also be easy to understand. Functions *firstkey*, *lastkey*, and *firstid* return the key of the first process on a list, the key of the last process on a list, or the *queuetab* index of the first process on a list. Usually, these functions are applied to nonempty lists, but they do not abort even if the list is empty because the *qkey* field is always initialized.

## 4.6 Basic Functions To Extract A Process From A List

Consider extracting a process from a list.† Recall that extracting an item from the head of a FIFO queue results in removing the item that has been in the queue the longest. For a priority queue, extracting from the head produces an item with highest priority. Similarly, extracting an item from the tail of the queue produces an item with lowest priority. As a result, we can construct three basic functions that are sufficient to handle extraction:

- *getfirst* — extract the process at the head of a list
- *getlast* — extract the process at the tail of a list
- *getitem* — extract a process at an arbitrary point

The code for the three basic functions can be found in file *getitem.c*.

†We will consider insertion into a list later.

```
/* getitem.c - getfirst, getlast, getitem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getfirst  -  Remove a process from the front of a queue
 *------------------------------------------------------------------------
 */
pid32   getfirst(
          qid16         q               /* ID of queue from which to    */
        )                               /* remove a process (assumed    */
                                        /* valid with no check)         */
{
        pid32   head;

        if (isempty(q)) {
                return EMPTY;
        }

        head = queuehead(q);
        return getitem(queuetab[head].qnext);
}

/*------------------------------------------------------------------------
 *  getlast  -  Remove a process from end of queue
 *------------------------------------------------------------------------
 */
pid32   getlast(
          qid16         q               /* ID of queue from which to    */
        )                               /* remove a process (assumed    */
                                        /* valid with no check)         */
{
        pid32 tail;

        if (isempty(q)) {
                return EMPTY;
        }

        tail = queuetail(q);
        return getitem(queuetab[tail].qprev);
}

/*------------------------------------------------------------------------
 *  getitem  -  Remove a process from an arbitrary point in a queue
 *------------------------------------------------------------------------
```

```
 */
pid32   getitem(
          pid32         pid             /* ID of process to remove      */
        )
{
        pid32   prev, next;

        next = queuetab[pid].qnext;     /* following node in list       */
        prev = queuetab[pid].qprev;     /* previous node in list        */
        queuetab[prev].qnext = next;
        queuetab[next].qprev = prev;
        return pid;
}
```

*Getfirst* takes a queue ID as an argument, verifies that the argument identifies a nonempty list, finds the process at the head of the list, and calls *getitem* to extract the process from the list. Similarly, *getlast* takes a queue ID as an argument, checks the argument, finds the process at the tail of the list, and calls *getitem* to extract the process. Each of the two functions returns the ID of the process that has been extracted.

*Getitem* takes a process ID as an argument, and extracts the process from the list in which the process is currently linked. Extraction consists of making the previous node point to the successor and the successor point to the previous node. Once a process has been unlinked from a list, *getitem* returns the process ID as the value of the function.

## 4.7 FIFO Queue Manipulation

We will see that many of the lists a process manager maintains consist of a *First-In-First-Out* (*FIFO*) queue. That is, a new item is inserted at the tail of the list, and an item is always removed from the head of the list. For example, a scheduler can use a FIFO queue to implement round-robin scheduling by placing the current process on the tail of a list and switching to the process on the head of the list.

Functions *enqueue* and *dequeue*, found in file *queue.c*, implement FIFO operations on a list. Because each list has both a head and tail, both insertion and extraction are efficient. For example, *enqueue* inserts a process just prior to the tail of a list, and *dequeue* extracts an item just after the head of the list. *Dequeue* takes a single argument that gives the ID of the list to use. *Enqueue* takes two arguments: the ID of the process to be inserted and the ID of a list on which to insert it.

```
/* queue.c - enqueue, dequeue */

#include <xinu.h>

struct qentry   queuetab[NQENT];        /* table of process queues      */

/*------------------------------------------------------------------------
 *  enqueue  -  Insert a process at the tail of a queue
 *------------------------------------------------------------------------
 */
pid32   enqueue(
          pid32         pid,            /* ID of process to insert      */
          qid16         q               /* ID of queue to use           */
        )
{
        int     tail, prev;             /* tail & previous node indexes */

        if (isbadqid(q) || isbadpid(pid)) {
                return SYSERR;
        }

        tail = queuetail(q);
        prev = queuetab[tail].qprev;

        queuetab[pid].qnext  = tail;    /* insert just before tail node */
        queuetab[pid].qprev  = prev;
        queuetab[prev].qnext = pid;
        queuetab[tail].qprev = pid;
        return pid;
}

/*------------------------------------------------------------------------
 *  dequeue  -  Remove and return the first process on a list
 *------------------------------------------------------------------------
 */
pid32   dequeue(
          qid16         q               /* ID queue to use              */
        )
{
        pid32   pid;                    /* ID of process removed        */

        if (isbadqid(q)) {
                return SYSERR;
        } else if (isempty(q)) {
                return EMPTY;
```

```
        }

        pid = getfirst(q);
        queuetab[pid].qprev = EMPTY;
        queuetab[pid].qnext = EMPTY;
        return pid;
}
```

Function *enqueue* calls *isbadpid* to check whether its argument is a valid process ID. The next chapter shows that *isbadpid* consists of an inline function that checks whether the ID is in the correct range and that a process with that ID exists.

File *queue.c* includes *xinu.h*, which includes the complete set of Xinu include files:

```
/* xinu.h - include all system header files */

#include <kernel.h>
#include <conf.h>
#include <process.h>
#include <queue.h>
#include <sched.h>
#include <semaphore.h>
#include <memory.h>
#include <bufpool.h>
#include <clock.h>
#include <mark.h>
#include <ports.h>
#include <uart.h>
#include <tty.h>
#include <device.h>
#include <interrupt.h>
#include <file.h>
#include <rfilesys.h>
#include <rdisksys.h>
#include <lfilesys.h>
#include <ag71xx.h>
#include <ether.h>
#include <mips.h>
#include <nvram.h>
#include <gpio.h>
#include <net.h>
#include <arp.h>
#include <udp.h>
#include <dhcp.h>
#include <icmp.h>
```

```
#include <name.h>
#include <shell.h>
#include <date.h>
#include <prototypes.h>
```

Combining the set of header files into a single include file helps programmers because it insures that all pertinent definitions are available and guarantees that the set of include files are processed in a valid sequence. Later chapters consider the contents of each include file.

## 4.8 Manipulation Of Priority Queues

A process manager often needs to select from a set of processes the process that has highest priority. Consequently, the linked list routines must be able to maintain lists of processes that each have an associated *priority*. In our example system, a priority is an integer value assigned to the process. In general, the task of examining a process with highest priority is performed frequently compared with the tasks of inserting and deleting processes. Thus, a data structure used to manage lists of processes should be designed to make finding the highest priority process efficient compared to insertion or deletion.

A variety of data structures has been devised to store a set of items that can be selected in priority order. Any such data structure is known generically as a *priority queue*. Our example system uses a linear list to store a priority queue where the priority of a process serves as a key in the list. Because the list is ordered in descending order by key, the highest priority process can always be found at the head of the list. Thus, finding the highest priority process takes constant time. Insertion is more expensive because the list must be searched to determine the location at which a new item should be inserted.

In a small embedded system where one only expects two or three processes to be on a given priority queue at any time, a linear list suffices. For a large system where many items appear in a given priority queue or where the number of insertions is high compared to the number of times items are extracted, a linear list can be inefficient. An exercise considers the point further.

Deletion from an ordered list is trivial: the first node is removed from the list. When an item is inserted, list order must be maintained. *Insert*, which is shown below, inserts a process on a list ordered by priority. The function takes three arguments: the ID of a process to be inserted, the ID of a queue on which to insert the process, and an integer priority for the process. *Insert* uses the *qkey* field in *queuetab* to store the process's priority. To find the correct location in the list, *insert* searches for an existing element with a key less than the key of the element being inserted. During the search, integer *curr* moves along the list. The loop must eventually terminate because the key

of the tail element contains a value less than the smallest valid key. Once the correct location has been found, *insert* changes the necessary pointers to link the new node into the list.

```
/* insert.c - insert */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  insert  -  Insert a process into a queue in descending key order
 *------------------------------------------------------------------------
 */
status  insert(
          pid32         pid,            /* ID of process to insert      */
          qid16         q,              /* ID of queue to use           */
          int32         key             /* key for the inserted process */
        )
{
        int16   curr;                   /* runs through items in a queue*/
        int16   prev;                   /* holds previous node index    */

        if (isbadqid(q) || isbadpid(pid)) {
                return SYSERR;
        }

        curr = firstid(q);
        while (queuetab[curr].qkey >= key) {
                curr = queuetab[curr].qnext;
        }

        /* insert process between curr node and previous node */

        prev = queuetab[curr].qprev;    /* get index of previous node   */
        queuetab[pid].qnext = curr;
        queuetab[pid].qprev = prev;
        queuetab[pid].qkey = key;
        queuetab[prev].qnext = pid;
        queuetab[curr].qprev = pid;
        return OK;
}
```

## 4.9 List Initialization

The procedures described above all assume that although it may be empty, a given queue has been initialized. We now consider the code that creates an empty list. It is appropriate that the code to create an empty list occurs at the end of the chapter because it brings up an important point about the design process:

Initialization is the final step in design.

It may seem strange to defer initialization because a designer cannot postpone thinking about initialization altogether. However, the general paradigm can be stated as follows: first, design the data structures needed when the system is running, and then figure out how to initialize the data structures. Partitioning the "steady state" aspect of the system from the "transient state" helps focus the designer's attention on the most important aspect, and avoids the temptation of sacrificing good design for easy initialization.

Initialization of entries in the *queuetab* structure is performed on demand as entries are needed. A running process calls function *newqueue* to create a new list. The system maintains a global pointer to the next unallocated element of *queuetab*, which makes it easy to allocate the list.

In theory, the head and tail of a list could be allocated from any of the unused entries in *queuetab*. In practice, however, choosing arbitrary locations would require a caller to store two items: the indices of the head and tail. To optimize storage, we make the rule that:

*The head and tail nodes for a list, X, are allocated from successive locations in the* queuetab *array, and list X is identified by the index of the head.*

In the code, *newqueue* allocates a pair of adjacent positions in the *queuetab* array to use as head and tail nodes, and initializes the list to empty by pointing the successor of the head to the tail and the predecessor of the tail to the head. *Newqueue* assigns the value *EMPTY* to unused pointers (i.e., the successor of the tail and the predecessor of the head). When it initializes a list, *newqueue* also sets the key fields in the head and the tail to the maximum and minimum integer values, respectively, with the assumption that neither value will be used as a key. Only one allocation function is needed because a list can be used to implement a FIFO queue or a priority queue.

Once it finishes with initialization, *newqueue* returns the index of the list head to its caller. The caller only needs to store one value because the ID of the tail can be computed by adding 1 to the ID of the head.

```
/* newqueue.c - newqueue */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  newqueue  -  Allocate and initialize a queue in the global queue table
 *------------------------------------------------------------------------
 */
qid16   newqueue(void)
{
        static qid16    nextqid=NPROC;/* next list in queuetab to use   */
        qid16           q;              /* ID of allocated queue        */

        q = nextqid;
        if (q > NQENT) {                /* check for table overflow     */
                return SYSERR;
        }

        nextqid += 2;                   /* increment index for next call*/

        /* initialize head and tail nodes to form an empty queue */

        queuetab[queuehead(q)].qnext = queuetail(q);
        queuetab[queuehead(q)].qprev = EMPTY;
        queuetab[queuehead(q)].qkey  = MAXKEY;
        queuetab[queuetail(q)].qnext = EMPTY;
        queuetab[queuetail(q)].qprev = queuehead(q);
        queuetab[queuetail(q)].qkey  = MINKEY;
        return q;
}
```

## 4.10 Perspective

Using a single data structure for process lists makes it possible to create general-purpose linked list manipulation functions, which reduce the size of the code by avoiding duplication. Using an implicit data structure with relative pointers reduces the memory used. For small embedded systems, compacting code and data is necessary. What about systems that have plenty of memory? Interestingly, a general principle applies: unless care is taken, successive generations of software expand to fill whatever memory is available. Thus, thinking carefully about a design is always important: there are never sufficient resources to justify wasteful inefficiency.

## 4.11 Summary

The chapter describes linked-list functions found in a process manager. In our example system, linked lists of processes are kept in a single, uniform data structure, the *queuetab* array. Functions that manipulate lists of processes can produce FIFO queues or priority queues. All lists have the same form: they are doubly linked, each has both a head and tail, and each node has an integer key field. Keys are used when the list is a priority queue; keys are ignored if the list is a FIFO queue.

To reduce the size required, the Xinu implementation uses relative pointers and an implicit data structure. A node in the data structure either corresponds to a process or to a list head or tail.

## EXERCISES

**4.1** In what sense is the queue structure described here an *implicit* data structure?

**4.2** If priority values range from –8 to +8, how many bits are required to store each key in *queuetab*?

**4.3** Create a separate set of functions that allows one to create a singly-linked list, and insert items in either FIFO or priority order. By how much does the second set of routines increase memory usage? Does having a separate set of routines decrease CPU usage? Explain.

**4.4** Does *insert* work correctly for all possible key values? If not, for which key(s) does it fail?

**4.5** Implement procedures to manipulate lists using pointers instead of indices into an array of structures. What is the difference in memory use and CPU time?

**4.6** Compare the complexity of functions like *isempty* implemented with pointers and with array indexing.

**4.7** Larger systems sometimes use a data structure known as a *heap* to contain a priority queue. What is a heap? Will its use be more or less expensive than an ordered, doubly linked list when the list size is between 1 and 3?

**4.8** Functions *getfirst*, *getlast*, and *getitem* do not check whether their argument is a valid queue ID. Modify the code to insert the appropriate checks.

**4.9** The code generated to convert a subscript into a memory address may use multiplication. Try padding the size of a *qentry* to a power of two bytes, and examine the resulting code to see if the compiler uses a shift instead of multiplication.

**4.10** In the previous exercise, measure a series of insertions and deletions to determine the difference in speed between the padded and unpadded versions of the data structure.

**4.11** If a structure contains data items that are not multiples of four bytes on an architecture with strict word alignment (such as MIPS), the code a compiler generates to access a structure member may include masking and shifting. Try altering the fields of qentry so that members are aligned on machine word boundaries, and examine the impact on the size of the queue table and the resulting code for accessing members.

**4.12** Modify *newqueue* to check for an error caused by attempting to allocate more than *NQENT* entries.

# NOTES

# Chapter Contents

# 5

# Scheduling and Context Switching

*What is called a sincere work is one that is endowed with enough strength to give reality to an illusion.*

— Max Jacob

## 5.1 Introduction

An operating system achieves the illusion of concurrent execution by rapidly switching a processor among several computations. Because the speed of the computation is extremely fast compared to that of a human, the effect is impressive — multiple activities appear to proceed simultaneously.

Context switching, which lies at the heart of the juggling act, consists of stopping the current process, saving enough information so it may be restarted later, and starting another process. What makes such a change difficult is that the CPU cannot be stopped during a context switch — the CPU must continue to execute the code that switches to a new process.

This chapter describes the basic context switching mechanism, showing how an operating system saves the state information from one process, chooses another process to run from among those that are ready, and relinquishes control to the new process. The chapter describes the data structure that holds information about processes that are not currently executing, and explains how the context switch uses the data structure. For the present, we will ignore the questions of when and why processes choose to switch context. Later chapters answer the questions, showing how higher levels of the operating system use the context switch described here.

## 5.2 The Process Table

An operating system keeps all information about processes in a data structure known as a *process table*. A process table contains one entry for each process that currently exists. We will see that a new entry must be allocated each time a process is created and an entry is removed when a process terminates. Because exactly one process is executing at any time, exactly one of the entries in a process table corresponds to an active process — the saved state information in the process table is out of date for the executing process. Each of the other process table entries contains information about a process that has been stopped temporarily. To switch context, the operating system saves information about the currently running process in its process table entry, and restores information from the process table entry corresponding to the process it is about to execute.

Exactly what information must be saved in the process table? The system must save any values that will be destroyed when the new process runs. Consider the stack. Because each process has its own separate stack memory, a copy of the entire stack need not be saved. However, when it executes, a process will change the hardware stack pointer register. Therefore, the contents of the stack pointer register must be saved when a process temporarily stops executing, and must be restored when the process resumes execution. Similarly, copies of other general-purpose registers must be saved and restored. In addition to values from the hardware, an operating system keeps meta-information in the process table. We will see how operating system functions use the meta-information for resource accounting, error prevention, and other administrative tasks. For example, the process table on a multi-user system stores the user ID of the user who owns the process. Similarly, if the operating system places policy limits on the memory that a process can allocate, the limit might be placed in the process table. The details of items in the process table will become clear in later chapters as we examine operating system functions that operate on processes.

The process table in our example system, *proctab*, consists of an array with *NPROC* entries. Each entry in *proctab* consists of a *procent* structure that defines the information kept for a process. Figure 5.1 lists key items found in a process table entry.

Throughout the operating system, each process is identified by an integer process ID. The following rule gives the relationship between process IDs and the process table:

> *A process is referenced by its process ID, which is the index of the proctab entry that contains the process's saved state information.*

| Field | Purpose |
|---|---|
| prstate | The current status of the process (e.g., whether the process is currently executing or waiting) |
| prprio | The scheduling priority of the process |
| prstkptr | The saved value of the process's stack pointer when the process is not executing |
| prstkbase | The address of the highest memory location in the memory region used as the process's stack |
| prstklen | A limit on the maximum size that the process's stack can grow |
| prname | A name assigned to the process that humans use to identify the process's purpose |

**Figure 5.1** Key items found in the Xinu process table.

As an example, consider how the code finds information about a process. The state information for a process with ID 3 can be found in proctab[3], and the state information for a process with ID 5 can be found in proctab[5]. Using the array index as an ID makes locating information efficient.

Each entry in *proctab* is defined to be a struct of type *procent*. The declaration of struct *procent* can be found in file *process.h* along with other declarations related to processes. Some fields in the process table contain information that the operating system needs to manage the process (e.g., the information needed to free the process's stack memory when the process completes). Other fields are used only for debugging. For example, field *prname* contains a character string that identifies the process; the field is not used except when a human tries to debug a problem.

```
/* process.h - isbadpid */

/* Maximum number of processes in the system */

#ifndef NPROC
#define NPROC           8
#endif

/* Process state constants */

#define PR_FREE         0       /* process table entry is unused        */
#define PR_CURR         1       /* process is currently running         */
#define PR_READY        2       /* process is on ready queue            */
#define PR_RECV         3       /* process waiting for message          */
#define PR_SLEEP        4       /* process is sleeping                  */
#define PR_SUSP         5       /* process is suspended                 */
#define PR_WAIT         6       /* process is on semaphore queue        */
#define PR_RECTIM       7       /* process is receiving with timeout    */

/* Miscellaneous process definitions */

#define PNMLEN          16      /* length of process "name"             */
#define NULLPROC        0       /* ID of the null process               */

/* Process initialization constants */

#define INITSTK         65536   /* initial process stack size           */
#define INITPRIO        20      /* initial process priority             */
#define INITRET         userret /* address to which process returns     */

/* Reschedule constants for ready */

#define RESCHED_YES     1       /* call to ready should reschedule      */
#define RESCHED_NO      0       /* call to ready should not reschedule  */

/* Inline code to check process ID (assumes interrupts are disabled)    */

#define isbadpid(x)     ( ((pid32)(x) < 0) || \
                          ((pid32)(x) >= NPROC) || \
                          (proctab[(x)].prstate == PR_FREE))

/* Number of device descriptors a process can have open */

#define NDESC           5       /* must be odd to make procent 4N bytes */
```

```
/* Definition of the process table (multiple of 32 bits) */

struct procent {                /* entry in the process table           */
        uint16  prstate;        /* process state: PR_CURR, etc.         */
        pri16   prprio;         /* process priority                     */
        char    *prstkptr;      /* saved stack pointer                  */
        char    *prstkbase;     /* base of run time stack               */
        uint32  prstklen;       /* stack length in bytes                */
        char    prname[PNMLEN]; /* process name                         */
        uint32  prsem;          /* semaphore on which process waits     */
        pid32   prparent;       /* id of the creating process           */
        umsg32  prmsg;          /* message sent to this process         */
        bool8   prhasmsg;       /* nonzero iff msg is valid             */
        int16   prdesc[NDESC];  /* device descriptors for process       */
};

/* Marker for the top of a process stack (used to help detect overflow) */
#define STACKMAGIC      0x0A0AAAA9

extern  struct  procent proctab[];
extern  int32   prcount;        /* currently active processes           */
extern  pid32   currpid;        /* currently executing process          */
```

## 5.3 Process States

To record exactly what each process is doing and to validate operations performed on the process, each process is assigned a *state*. An operating system designer defines the set of possible states as the design proceeds. Because many of the system functions that operate on processes use the state to determine whether an operation is valid, the set of states must be completely defined before the system can be implemented.

Xinu uses field *prstate* in the process table to record state information for each process. The system defines seven valid states and a symbolic constant for each. The system also defines an additional constant that is assigned when a given table entry is unused (i.e., no process has been created to use that particular entry). File *process.h* contains the definitions; Figure 5.2 lists the symbolic state constants and the meaning of each.

Because it runs as an embedded system, Xinu keeps the code and data for all processes in memory at all times. In larger operating systems, where a process executes an application program, the system can move a process to secondary storage when the process is not currently executing. Thus, in those systems, the process state also determines whether the process must reside in memory or can be moved to disk temporarily.

| Constant | Meaning |
|---|---|
| PR_FREE | The entry in the process table is unused (not really a process state) |
| PR_CURR | The process is currently executing |
| PR_READY | The process is ready to execute |
| PR_RECV | The process is waiting for a message |
| PR_SLEEP | The process is waiting for a timer |
| PR_SUSP | The process is suspended |
| PR_WAIT | The process is waiting on a semaphore |
| PR_RECTIM | The process is waiting for a timer or a message, whichever occurs first |

**Figure 5.2** The seven symbolic constants that can be assigned to the state of a process.

## 5.4 Ready And Current States

Later chapters will explain each process state in detail, and show how and why system functions change a process's state. The remainder of this chapter focuses on the current and ready states.

Almost every operating system includes *ready* and *current* process states. A process is classified *ready* if the process is ready (i.e., eligible) for CPU service but is not currently executing; the single process receiving CPU service is classified as *current*.

## 5.5 A Scheduling Policy

Switching from the currently executing process to another process consists of two steps: selecting a process from among those that are eligible to use the CPU, and giving control of the CPU to the selected process. Software that implements the policy for selecting a process is called a *scheduler*. In Xinu, function *resched* makes the selection according to the following well-known scheduling policy:

> *At any time, the highest priority process eligible for CPU service is executing. Among processes with equal priority scheduling is round-robin.*

Two aspects of the policy deserve attention:

- The currently executing process is included in the set of eligible processes. Thus, if process *p* is currently executing and has a higher priority than any of the other processes, process *p* will continue to execute.
- The term *round-robin* refers to a situation in which a set of *k* processes all have the same priority and the priority of processes in the set is higher than the priority of other processes. The round-robin policy means that members of the set will receive service one after another so all members of the set have an opportunity to execute before any member has a second opportunity.

## 5.6 Implementation Of Scheduling

The key to understanding a scheduler lies in knowing that a scheduler is merely a function. That is, the operating system scheduler is not an active agent that picks up the CPU from one process and moves it to another. Instead, a running process invokes the scheduler:†

*A scheduler consists of a function that a running process calls to willingly give up the CPU.*

Recall that a process priority consists of a positive integer, and the priority for a given process is stored in the *prprio* field of the process's entry in the process table. A user assigns a priority to each process to control how the process will be selected for CPU service. A variety of complex scheduling policies have been proposed and measured, including schedulers that adjust the priority of processes dynamically, based on the observed behavior of each process. For most embedded systems, however, process priorities remain relatively static (typically, the priority does not change after a process has been created).

To make the selection of a new process fast, our example system stores all ready processes on a list known as a *ready list*. Processes on a ready list are stored in descending order by priority. Thus, a highest priority process is immediately accessible at the head of the list.

In the example code, the ready list is stored in the *queuetab* array described in Chapter 4, and the scheduler uses functions from Chapter 4 to update and access the list. That is, the key in each element on the ready list consists of the priority for the process to which the element corresponds. Global variable *readylist* contains the queue ID for the ready list.

†Later chapters explain how and why a process invokes the scheduler.

Should the current process be kept on the ready list? The answer depends on details of the implementation. Either approach is feasible provided an entire system follows one approach or the other. Xinu implements the following policy:

> *The current process does not appear on the ready list; to provide fast access to the current process, its ID is stored in global integer variable* currpid.

Consider what happens when the CPU switches from one process to another. The currently executing process relinquishes the CPU. Often, the process that was executing remains eligible to use the CPU again. In such situations, the scheduler must change the state of the current process to *PR_READY* and insert the process onto the ready list, insuring that it will be considered for CPU service again later. In other cases, however, the current process does not remain ready to execute, which means the process should not be placed on the ready list.

How does the scheduler decide whether to move the current process onto the ready list? In Xinu, the scheduler does not receive an explicit argument that specifies the disposition of the current process. Instead, the system functions use an implicit argument: if the current process should not remain ready, before calling *resched*, the current process's *prstate* field must be set to the desired next state. Whenever it prepares to switch to a new process, *resched* checks the *prstate* field of the current process. If the state still specifies *PR_CURR*, *resched* assumes the process should remain ready, and moves the process to the ready list. Otherwise, *resched* assumes the next state has already been chosen. The next chapter shows an example.

In addition to moving the current process to the ready list, *resched* completes every detail of scheduling and context switching except saving and restoring machine registers (which cannot be done directly in a high-level language like C). *Resched* selects a new process to run, updates the process table entry for the new process, removes the new process from the ready list, marks it current, and updates *currpid*. It also resets the preemption counter, something we will consider later. Finally, *resched* calls function *ctxsw* to save the hardware registers of the current process and restore the registers for the new process. The code can be found in file *resched.c*:

```
/* resched.c - resched */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  resched  -  Reschedule processor to highest priority eligible process
 *------------------------------------------------------------------------
 */
void    resched(void)           /* assumes interrupts are disabled      */
{
```

```
        struct procent *ptold;  /* ptr to table entry for old process   */
        struct procent *ptnew;  /* ptr to table entry for new process   */

        /* If rescheduling is deferred, record attempt and return */

        if (Defer.ndefers > 0) {
                Defer.attempt = TRUE;
                return;
        }

        /* Point to process table entry for the current (old) process */

        ptold = &proctab[currpid];

        if (ptold->prstate == PR_CURR) {  /* process remains running */
                if (ptold->prprio > firstkey(readylist)) {
                        return;
                }

                /* Old process will no longer remain current */

                ptold->prstate = PR_READY;
                insert(currpid, readylist, ptold->prprio);
        }

        /* Force context switch to highest priority ready process */

        currpid = dequeue(readylist);
        ptnew = &proctab[currpid];
        ptnew->prstate = PR_CURR;
        preempt = QUANTUM;              /* reset time slice for process */
        ctxsw(&ptold->prstkptr, &ptnew->prstkptr);

        /* Old process returns here when resumed */

        return;
}
```

*Resched* begins by checking global variable *Defer.ndefers* to see whether rescheduling is deferred. If so, *resched* sets global variable *Defer.attempt* to indicate that an attempt was made during the deferral period and returns to the caller. Deferred rescheduling is provided to accommodate device drivers for hardware that requires a driver to service multiple devices on a single interrupt. For now, it is sufficient to understand that rescheduling can be deferred temporarily.

Once it passes the test for deferral, *resched* examines an implicit parameter: the state of the current process. If the state variable contains *PR_CURR* and the current process's priority is the highest in the system, *resched* returns and the current process remains running. If the state specifies that the current process should remain eligible to use the CPU but the current process does not have the highest priority, *resched* adds the current process to the ready list. *Resched* then removes the process at the head of the ready list (the highest priority process), and performs a context switch.

It may be difficult to envision how context switching occurs because each concurrent process has its own instruction pointer. To see how concurrency operates, suppose that process $P_1$ is running and calls *resched*. If *resched* chooses to switch to process $P_2$, process $P_1$ will be stopped in the call to *ctxsw*. However, process $P_2$ is free to run, and can execute arbitrary code. At a later time when *resched* switches back to process $P_1$, execution will resume where it left off — in the call to *ctxsw*. The location at which $P_1$ is executing does not change just because $P_2$ used the CPU. When process $P_1$ runs, the call to *ctxsw* will return to *resched*. A later section considers additional details.

## 5.7 Implementation Of Context Switching

Because registers and hardware status cannot be manipulated directly with a high-level language, *resched* calls an assembly language function, *ctxsw*, to switch context from one process to another. The code for *ctxsw* is, of course, machine dependent. The last step consists of resetting the program counter (i.e., jumping to the location in the new process at which execution should resume). In Xinu, the text segment for the new process will be present in memory because Xinu keeps all parts of the program resident. The point is that the operating system must load all other state variables for the new process before jumping to the new process. Some architectures contain two atomic instructions used in context switching: one that stores processor state information in successive memory locations and another that loads processor state from successive memory locations. On such architectures, context switching code executes a single instruction to save the processor state on the current process's stack and another instruction to load the processor state for the new process. Of course, each instruction takes many cycles. RISC architectures implement *ctxsw* with a long sequence of instructions, but each instruction executes quickly.

## 5.8 State Saved In Memory

To understand how *ctxsw* saves processor states, imagine that we can look at the memory of a system that has three active processes: two of which are ready and one of which is currently executing. Each process has a stack. The process that is currently executing is using its stack. When it calls a function, the executing process must allocate space on the stack for local variables and argument storage. When it returns from a function, the variables are popped off the stack. If the context switching function is

designed to store a copy of the machine state for a process on the process's stack, we will find that when they were running, the other two processes each pushed values on their stack because they each performed a context switch when they were last executing. Figure 5.3 illustrates the configuration.

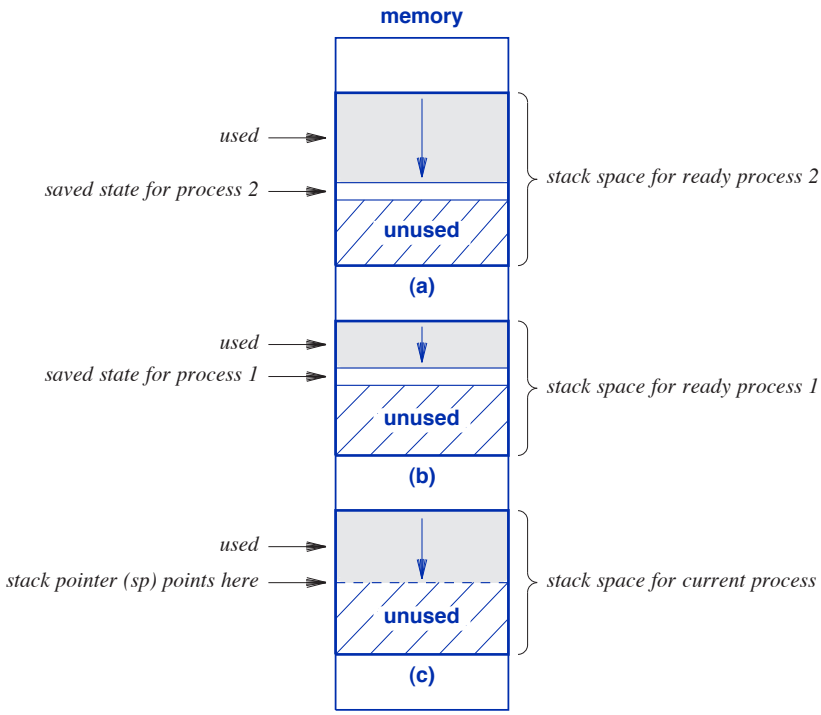



**Figure 5.3** Illustration of stacks in memory for (a) and (b) processes on the ready list, and (c) a process that is currently executing.

## 5.9 Context Switch On A MIPS Processor

On a RISC architecture, such as MIPS, each instruction can store only one register. Thus, N instructions must be executed to save N registers. Like most operating systems, Xinu follows the convention of saving state on a process's stack. As a consequence, our MIPS version of *ctxsw* performs four basic steps:

- Execute a sequence of instructions that pushes the contents of the processor registers on the stack of the process that is running when *ctxsw* is called.
- Save the stack pointer in the process table entry for the current process, and load the stack pointer for the "new" process.

- Execute a sequence of instructions that reload the processor registers from values previously saved on the new process's stack.
- Jump to the location in the new process at which execution should resume.

Because context switching involves the direct manipulation of processor registers, the code is written in assembly language. To save the contents of processor registers, *ctxsw* allocates *CONTEXT* bytes on the stack; the allocated area is sufficient to hold all the saved register values. *Ctxsw* then stores register values in successive words of the allocated region.

The second step, saving and restoring stack pointers, is straightforward, and only requires two instructions. *Ctxsw* receives two pointers as arguments: the address of the location in the process table where the current process stack pointer should be stored and the address in the process table that contains the new stack pointer. Thus, a single instruction stores the stack pointer at the location given by the first argument, and another loads the stack pointer from the address given by the second argument.

After the second step, the stack pointer points to the new process's stack. *Ctxsw* extracts the set of values that was saved on the stack for the process, and loads the values into processor registers. Once the values have been loaded, *ctxsw* removes *CONTEXT* bytes from the stack. File *ctxsw.S* contains the code.

```
/* ctxsw.S - ctxsw */

#include <mips.h>

.text
        .align 4
        .globl  ctxsw

/*------------------------------------------------------------------------
 *  ctxsw  -  Switch from one process context to another
 *------------------------------------------------------------------------
 */
        .ent ctxsw
ctxsw:
        /* Build context record on the current process' stack */

        addiu   sp, sp, -CONTEXT
        sw      ra, CONTEXT-4(sp)
        sw      ra, CONTEXT-8(sp)

        /* Save callee-save (non-volatile) registers */

        sw      s0, S0_CON(sp)
```

```
sw      s1, S1_CON(sp)
sw      s2, S2_CON(sp)
sw      s3, S3_CON(sp)
sw      s4, S4_CON(sp)
sw      s5, S5_CON(sp)
sw      s6, S6_CON(sp)
sw      s7, S7_CON(sp)
sw      s8, S8_CON(sp)
sw      s9, S9_CON(sp)

/* Save outgoing process' stack pointer */

sw      sp, 0(a0)

/* Load incoming process' stack pointer */

lw      sp, 0(a1)

/* At this point, we have switched from the run-time stack */
/*    of the outgoing process to the incoming process      */

/* Restore callee-save (non-volatile) registers from new stack  */

lw      s0, S0_CON(sp)
lw      s1, S1_CON(sp)
lw      s2, S2_CON(sp)
lw      s3, S3_CON(sp)
lw      s4, S4_CON(sp)
lw      s5, S5_CON(sp)
lw      s6, S6_CON(sp)
lw      s7, S7_CON(sp)
lw      s8, S8_CON(sp)
lw      s9, S9_CON(sp)

/* Restore argument registers for the new process */

lw      a0, CONTEXT(sp)
lw      a1, CONTEXT+4(sp)
lw      a2, CONTEXT+8(sp)
lw      a3, CONTEXT+12(sp)

/* Remove context record from the new process' stack */

lw      v0, CONTEXT-4(sp)
lw      ra, CONTEXT-8(sp)
```

```
        addiu   sp, sp, CONTEXT

        /* If this is a newly created process, ensure */
        /*    it starts with interrupts enabled       */

        beq     v0, ra, ctxdone
        mfc0    v1, CP0_STATUS
        ori     v1, v1, STATUS_IE
        mtc0    v1, CP0_STATUS

ctxdone:
        jr      v0
        .end ctxsw
```

As comments in the code suggest, a newly created process should execute with interrupts enabled; other processes will return from the call to context switch and execute *resched* with interrupts disabled. Because the MIPS design uses a co-processor to handle interrupts, the context switch must use the co-processor to enable interrupts explicitly. Our implementation uses a trick: when a process is created, the values stored at locations *CONTEXT–4(sp)* and *CONTEXT–8(sp)* differ, and for other processes, they are the same. As the last step, *ctxsw* compares the two values and enables interrupts if they differ.

## 5.10 An Address At Which To Restart A Process

A potential problem arises in context switching: the processor continues to execute during the context switch, which means registers can change. Thus, the code must be written carefully to insure that once a given register has been saved, the register is not changed (or the value will be lost when the process restarts). The program counter represents a special dilemma because storing the value means that when the process restarts, execution will continue at exactly the point in the code at which the instruction pointer was stored. If the context switch has not completed when the instruction pointer is stored, the process will restart at a point *before* the context switch has occurred. The code in *ctxsw* reveals how to resolve the situation: instead of saving the value of the program counter while *ctxsw* is running, choose an address at which the process should resume when restarted.

To understand how an appropriate address is chosen, think of an executing process. The process has called *resched* which has then called *ctxsw*. Instead of trying to save the program counter at a point in *ctxsw*, the code saves a value as if the process had just returned from the call to *ctxsw*. That is, the saved value of the program counter is taken from the return address, the address to which *ctxsw* would return if it were a normal procedure. Consequently:

*When a process restarts, the process begins executing in* resched *immediately following the call to* ctxsw.

All processes call *resched* to perform context switching, and *resched* calls *ctxsw*. Thus, if one were to freeze the system at an arbitrary instant and examine memory, the saved program counter for each ready process will have the same value — an address just after the call to *ctxsw* in *resched*. However, each process has its own stack of function calls, which means that when a given process resumes execution and returns from *resched*, the return may go to a different caller than the return in another process.

The notion of function return forms a key ingredient in keeping the system design clean. Function calls proceed downward through each level of the system, and each call returns. To enforce the design at all levels, the scheduler, *resched*, and the context switch, *ctxsw*, have each been designed to behave like any other procedure and return. To summarize:

*In Xinu, each function, including the scheduler and context switch, eventually returns to its caller.*

Of course, rescheduling allows other processes to execute, and the execution may take arbitrarily long (depending on process priorities). Thus, a considerable delay may elapse between a call to *resched* and the time the call returns and the process runs again.

## 5.11 Concurrent Execution And A Null Process

The concurrent execution abstraction is complete and absolute. That is, an operating system views all computation as part of a process — there is no way that the CPU can temporarily stop executing processes and execute a separate piece of code. The scheduler design reflects the following principle: the scheduler's only function is to switch the processor among the set of processes that are current or ready. The scheduler cannot execute any code that is not part of a process, and cannot create a new process. Figure 5.4 illustrates the possible state transitions.

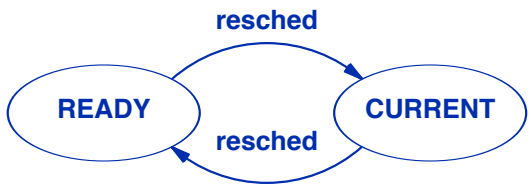



**Figure 5.4** Illustrations of state transitions for processes between the ready and current states.

We will see that a given process does not always remain ready to execute. For example, a process stops executing when it waits for I/O to complete or when it needs to use a shared resource that is already in use. What happens if all processes wait for I/O? *Resched* will fail because the code has been designed to assume at least one process will be eligible to execute at any time. When the currently executing process blocks, *resched* removes the first process from the ready list without verifying that the list is nonempty. If the list is empty, an error results. To summarize:

> *Because an operating system can only switch the CPU from one process to another, at least one process must remain ready to execute at all times.*

To insure that at least one process always remains ready to execute, Xinu uses a standard technique: it creates an extra process, called the *null process*, when the system boots. The null process has process ID zero and priority zero (a lower priority than any other process). The null process code, which will be shown in Chapter 22, consists of an infinite loop. Because all other processes must have a priority greater than zero, the scheduler switches to the null process only when no other process remains ready to run. In essence, the operating system switches the CPU to the null process when all other processes are blocked (e.g., waiting for I/O).†

## 5.12 Making A Process Ready And The Scheduling Invariant

When *resched* needs to move the current process onto the ready list, it manipulates the list directly. Making a process eligible for CPU service occurs so frequently that we designed a function to do just that. The function is named *ready*.

*Ready* takes two arguments: a process ID and a Boolean argument that controls whether *resched* should be called. Calls to *ready* use symbolic constants *RESCHED_YES* and *RESCHED_NO* to make the purpose of the argument clear in calls.

Consider the second argument. Our scheduling policy specifies that at any time, the highest priority eligible process must be executing. Thus, if it places a high priority process on the ready list, *ready* should call *resched* to insure that the policy is followed. We say that each operating system function should maintain a *scheduling invariant*: a function assumes the highest priority process was executing when the function was called, and must insure that the highest priority process is executing when the function returns. Thus, if a function changes the state of processes, the function must call *resched* to reestablish the invariant. *Ready* is an exception. To understand why, it is essential to know that some operating system functions move multiple processes onto the ready list (e.g., multiple timer events can occur at the same time). Rescheduling in the midst of such a move can result in incomplete operations. The solution consists of temporarily suspending the scheduling policy and allowing multiple calls of *ready* with argument *RESCHED_NO*. Once the entire set of processes has been added, a single

†Some CPUs include a special instruction that can be placed in a null process that stops the CPU until an interrupt occurs; using the special instruction may reduce the energy that the CPU consumes.

call to *resched* must be made to reinstate the policy and assure that the highest priority ready process is executing. We will see an example use of *ready* in Chapter 7. Meanwhile, it is sufficient to see how *ready* avoids rescheduling if the second argument is *RESCHED_NO*. File *ready.c* contains the code.

```
/* ready.c - ready */

#include <xinu.h>

qid16   readylist;                      /* index of ready list          */

/*------------------------------------------------------------------------
 *  ready  -  Make a process eligible for CPU service
 *------------------------------------------------------------------------
 */
status  ready(
          pid32         pid,            /* ID of process to make ready  */
          bool8         resch           /* reschedule afterward?        */
        )
{
        register struct procent *prptr;

        if (isbadpid(pid)) {
                return(SYSERR);
        }

        /* Set process state to indicate ready and add to ready list */

        prptr = &proctab[pid];
        prptr->prstate = PR_READY;
        insert(pid, readylist, prptr->prprio);

        if (resch == RESCHED_YES) {
                resched();
        }
        return OK;
}
```

## 5.13 Deferred Rescheduling

Recall that *resched* uses global variable *Defer.ndefers* to determine whether rescheduling is currently *deferred*.† Function *sched_cntl* provides an interface that is used to control deferral. A caller invokes:

†The code for *resched* can be found on page 74.

sched_cntl(DEFER_START);

to defer rescheduling, and

sched_cntl(DEFER_STOP);

to end a deferral period. To accommodate systems in which interrupt processing can be interrupted by a higher priority device, global variable *Defer.ndefers* acts as a reference count that is incremented whenever a process requests deferral to start and decremented whenever a process ends its deferral period. When the count reaches zero, *sched_cntl* tests global variable *Defer.attempt* to see if *resched* was called during the deferral period. If so, *sched_cntl* invokes *resched* before returning to its caller. File *sched.h* defines constants and variables used for deferral.

```
/* sched.h */

/* Constants and variables related to deferred rescheduling */

#define DEFER_START     1       /* start deferred rescehduling          */
#define DEFER_STOP      2       /* stop  deferred rescehduling          */

/* Structure that collects items related to deferred rescheduling       */

struct  defer   {
        int32   ndefers;        /* number of outstanding defers         */
        bool8   attempt;        /* was resched called during the        */
                                /*   deferral period?                   */
};

extern  struct  defer   Defer;
```

File *sched_cntl.c* contains the code to control deferral.

```
/* sched_cntl.c - sched_cntl */

#include <xinu.h>

struct  defer   Defer;

/*------------------------------------------------------------------------
 *  sched_cntl  -  control whether rescheduling is deferred or allowed
 *------------------------------------------------------------------------
 */
status  sched_cntl(             /* assumes interrupts are disabled      */
          int32 def             /* either DEFER_START or DEFER_STOP     */
        )
```

```
{

        switch (def) {

            /* Process request to defer:                                */
            /*  1) Increment count of outstanding deferral requests     */
            /*  2) If this is the start of deferral, initialize Boolean */
            /*          to indicate whether resched has been called     */

            case DEFER_START:
                if (Defer.ndefers++ == 0) {     /* increment deferrals  */
                        Defer.attempt = FALSE;  /* no attempts so far   */
                }
                return OK;

            /* Process request to stop deferring:                       */
            /*  1) Decrement count of outstanding deferral requests     */
            /*  2) If last deferral ends, make up for any calls to      */
            /*          resched that were missed during the deferral    */

            case DEFER_STOP:
                if (Defer.ndefers <= 0) {       /* none outstanding */
                        return SYSERR;
                }
                if (--Defer.ndefers == 0) {     /* end deferral period  */
                        if (Defer.attempt) {    /* resched was called   */
                                resched();      /*   during deferral    */
                        }
                }
                return OK;

            default:
                return SYSERR;
        }
}
```

## 5.14 Other Process Scheduling Algorithms

Process scheduling was once an important topic in operating systems, and many scheduling algorithms have been proposed as alternatives to the round-robin scheduler in Xinu. For example, one policy measures the amount of I/O a process performs and gives the CPU to the process that spends the most time doing I/O. Proponents of the policy argue that because I/O devices are slower than processors, choosing a process that performs I/O will increase the total throughput of the system.

Because scheduling is confined to a few functions, it is easy to experiment with the scheduling policy in Xinu. Changing *resched* and *ready* changes the basic scheduler. Of course, if the new policy uses data that Xinu does not already gather (e.g., the amount of time spent doing I/O), other functions may need to change to record the appropriate data.

## 5.15 Perspective

The most interesting aspect of scheduling and context switching arises because they are embedded as part of normal computation. That is, instead of the operating system implemented separately from the processes it controls, the operating system code is executed by the processes themselves. Thus, the system does not have an extra process that can stop the processor from executing one application and move it to another, scheduling and context switching occur as the side effect of a function call.

We will see that using processes to execute operating system code affects the design. When a programmer writes an operating system function, the programmer must accommodate execution by concurrent processes. Similarly, using processes to execute operating system code affects how the system interacts with I/O devices and how it handles interrupts.

## 5.16 Summary

Scheduling and context switching form a foundation for concurrent execution. Scheduling consists of choosing a process from among those that are eligible for execution. Context switching consists of stopping one process and starting a new one. To keep track of processes, the system uses a global data structure called a process table. Whenever it temporarily suspends a process, the context switch saves the processor state for the process on the process's stack and places a pointer to the stack in the process table. To restart a process, the context switch reloads the processor state information from the process's stack, and resumes execution in the process at the point the call to the context switch function returns.

To allow functions to determine when an operation is permitted, each process is assigned a *state*. A process that is using the CPU is assigned the *current* state, and a process that is eligible to use the CPU, but is not currently executing, is assigned the *ready* state. Because at least one process must remain eligible to execute at any time, the operating system creates an extra process at startup known as the *null process*. The null process has priority zero, and all other processes have priority greater than zero. Consequently, the null process only runs when no other process is eligible.

The chapter presents three functions that perform transitions between the *current* and *ready* states. Function *resched* performs scheduling, function *ctxsw* performs context switching, and function *ready* makes a process eligible to execute.

## EXERCISES

**5.1** If the operating system contains a total of $N$ processes, how many processes can be on the ready list at a given time? Explain.

**5.2** How do operating system functions know which process is executing at a given time?

**5.3** Rewrite *resched* to have an explicit parameter giving the disposition of the currently executing process, and examine the assembly code generated to determine the number of instructions executed in each case.

**5.4** What are the basic steps performed during a context switch?

**5.5** Investigate another hardware architecture (e.g., the Intel x86), and determine what information needs to be saved during a context switch.

**5.6** How much memory is needed to store processor state for a MIPS processor? Which registers must be saved, and why? How does the standard calling convention for a processor affect the answer?

**5.7** Suppose process $k$ has been placed on the ready list. When process $k$ becomes current, where will execution start?

**5.8** Why is a null process needed?

**5.9** Consider a modification to the code that stores processor state in the process table instead of on the process's stack (i.e., assume the process table entry contains an array that holds the contents of registers). What are the advantages of each approach?

**5.10** In the previous exercise, does saving registers in the process table reduce or increase the number of instructions executed during a context switch?

**5.11** Devise a scheduling policy for a dual-core processor (i.e., a processor that contains two separate CPUs that can execute in parallel).

**5.12** Extend the previous exercise: show that executing *resched* on one core may require changing the process that is running on the other core. (Note: many operating systems for dual-core processors avoid the problem by specifying that all operating system functions, including scheduling, run on one of the two cores.)

**5.13** The code contains two mechanisms used to defer rescheduling: *ready* has an argument that allows the caller to avoid rescheduling and *sched_cntl* sets a global bit that makes *resched* ignore calls. Why are both mechanisms included? Hint: compare the efficiency and overhead.

# Chapter Contents

# 6

# More Process Management

*When men willingly suspend fear, science flourishes.*

— Anonymous

## 6.1 Introduction

Chapter 5 discusses the concurrent execution abstraction and processes of execution. The chapter explains how an operating system stores information about processes in a table, and how each process is assigned a state. Chapter 5 also explains the concepts of scheduling and context switching. It shows how a scheduler implements a scheduling policy, and explains how a process moves between the ready and current states.

This chapter extends our study of process management. The chapter explains how a new process comes into existence, and what happens when a process exits. The chapter also examines a process state that allows a process to be suspended temporarily, and explores functions that move processes among the current, ready, and suspended states.

## 6.2 Process Suspension And Resumption

We will see that operating system functions sometimes need to temporarily stop a process from executing and at a later time resume execution. We say that a stopped process has been placed in a state of "suspended animation." Suspended animation can be used, for example, when a process waits for one of several restart conditions without knowing which will occur first.

The first step in implementing operating system functionality consists of defining a set of operations. In the case of suspended animation, only two conceptual operations provide all the functionality that is needed:

- *Suspend* stops a process and places the process in suspended animation (i.e., makes the process ineligible to use the CPU).
- *Resume* continues execution of a previously suspended process (i.e., makes the process eligible to use the CPU again).

Because it is not eligible to use the CPU, a suspended process cannot remain in either the *ready* or *current* states. Thus, a new state must be invented. We call the new state *suspended*, and add the new state and associated transitions to the state diagram. Figure 6.1 shows the extended state diagram, which summarizes how *suspend* and *resume* affect the state. The resulting diagram documents the possible transitions among the *ready*, *current*, and *suspended* states.

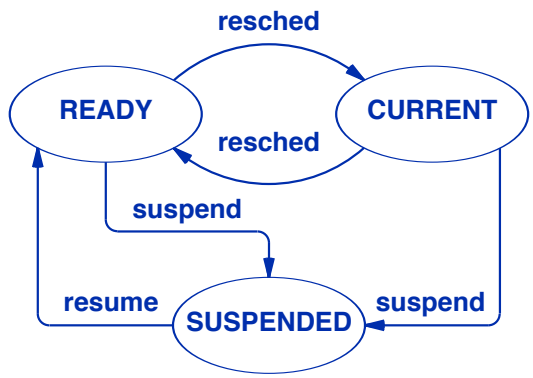



**Figure 6.1** Transitions among the current, ready, and suspended states.

## 6.3 Self-Suspension And Information Hiding

Although each state transition in Figure 6.1 has a label that specifies a particular function, process suspension differs from scheduling in a significant way: instead of acting on the current process, *suspend* allows one process to suspend another process. More important, because a suspended process cannot resume itself, *resume* must allow an executing process to resume a previously suspended process. Thus, *suspend* and *resume* each take an argument that specifies the ID of a process on which the operation should be performed.

Can a process suspend itself? Yes. To do so, a process must obtain its process ID and then pass the ID as an argument to *suspend*. An implementation may seem obvious. Because global variable *currpid* contains the process ID of the currently executing process, a self-suspension can be achieved with:

```
suspend( currpid );
```

However, a well-designed operating system design adheres to the *principle of information hiding*: implementation details are not generally revealed. Thus, instead of permitting processes to access global variables like *currpid* directly, Xinu includes a function named *getpid* that a process can call to obtain its ID. Thus, to suspend itself, a process calls:

```
suspend( getpid( ) );
```

The present implementation of *getpid* merely returns the value of *currpid*, which may seem like unnecessary overhead. However, the advantage of information hiding becomes clear when one considers modifying the operating system. If all processes call *getpid*, a designer can change the details of where and how the current process ID is stored without changing other code.

The point is:

> *A good system design follows the principle of information hiding, which states that implementation details are not revealed unless necessary. Hiding such details makes it possible to change the implementation of a function without rewriting code that uses the function.*

## 6.4 The Concept Of A System Call

In theory, process resumption is straightforward. The process must be placed in the ready state and inserted in the correct position on the ready list. The *ready* function described in the previous chapter performs both tasks, so it may seem that *resume* is unnecessary. In practice, however, *resume* adds an extra layer of protection: it makes no assumptions about the caller or the correctness of the arguments. In particular, an arbitrary process can invoke *resume* at an arbitrary time with arbitrary arguments.

We used the term *system call* to distinguish a function like *resume* from internal functions like *ready*. In general, we think of the set of system calls as defining a view of the operating system from the outside — application processes invoke system calls to obtain services. In addition to adding a layer of protection, the system call interface provides another example of information hiding: application processes remain unaware of the internal implementation, and can only use the set of system calls to obtain service. We will see that the distinction between system calls and other functions appears throughout an operating system. To summarize:

> *System calls, which define operating system services for applications, protect the system from illegal use and hide information about the underlying implementation.*

To provide protection, system calls handle three aspects of computation that underlying functions do not:

- Check all arguments
- Insure that changes leave global data structures in a consistent state
- Report success or failure to the caller

In essence, a system call cannot make any assumptions about the process that is making the call. Thus, instead of assuming that the caller has supplied correct and meaningful argument values, a system call checks each argument. More important, many system calls make changes to operating system data structures, such as the process table and lists of processes stored in the queue structure. A system call must guarantee that no other process will attempt to change data structures at the same time, or inconsistencies can result. Because it cannot make assumptions about the conditions under which it will be invoked, a system call must take steps to prevent other processes from executing concurrently while data structures are being changed. There are two aspects:

- Avoid invoking any functions that voluntarily relinquish the CPU
- Disable interrupts to prevent involuntarily relinquishing the CPU

To prevent voluntarily relinquishing the CPU, a system call must avoid direct or indirect calls to *resched*. That is, while changes are in progress, a system call cannot invoke *resched* directly and cannot invoke any function that calls *resched*. To prevent involuntarily relinquishing the CPU, a system call disables interrupts until a change is complete. In Chapter 12, we will understand the reason: hardware interrupts can result in rescheduling because some interrupt routines call *resched*.

An example system call will help clarify the two aspects. Consider the code for *resume* that is contained in file *resume.c*.

```
/* resume.c - resume */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  resume  -  Unsuspend a process, making it ready
 *------------------------------------------------------------------------
 */
pri16   resume(
          pid32         pid             /* ID of process to unsuspend   */
        )
{
```

```
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        pri16   prio;                   /* priority to return           */

        mask = disable();
        prptr = &proctab[pid];
        if (isbadpid(pid) || (prptr->prstate != PR_SUSP)) {
                restore(mask);
                return (pri16)SYSERR;
        }
        prio = prptr->prprio;           /* record priority to return    */
        ready(pid, RESCHED_YES);
        restore(mask);
        return prio;
}
```

## 6.5 Interrupt Control With Disable And Restore

As expected, the code in *resume* checks argument *pid* to insure that the caller has supplied a valid process ID and the specified process is in the suspended state. Before it performs any computation, however, *resume* guarantees that no interrupts will occur (i.e., no context switching can occur until *resume* invokes an operating system function that causes a context switch). To control interrupts, *resume* uses a pair of functions:†

- Function *disable* disables interrupts and returns the previous interrupt status to its caller.
- Function *restore* reloads an interrupt status from a previously saved value.

As expected, *resume* disables interrupts immediately upon entry. *Resume* can return in two ways: because an error is detected or because *resume* finishes the requested operation successfully. In either case, *resume* must call *restore* before returning to reset the interrupt status to the same value the caller was using when the call began.

Programmers who do not have experience writing code for operating systems often expect a system call to enable interrupts before returning. However, *restore* provides more generality. To see why, observe that because it restores interrupts rather than simply enabling them, *resume* works correctly whether it is called with interrupts enabled or disabled. On the one hand, if a function has interrupts disabled when it calls *resume*, the call will return with interrupts disabled. On the other hand, if a function has interrupts enabled when it calls *resume*, the call will return with interrupts enabled.

†Chapter 12 explains the details of interrupt handling.

*System calls must disable interrupts to prevent other processes from changing global data structures; using a disable/restore paradigm increases generality.*

## 6.6 A System Call Template

Another way to look at interrupt handling focuses on an invariant that a system function must maintain:

*An operating system function always returns to its caller with the same interrupt status as when it was called.*

To insure the invariant is maintained, operating system functions follow the general approach that Figure 6.2 illustrates.

```
syscall function_name(args)  {
        intmask mask;               /* interrupt mask                 */
        mask = disable();           /* disable interrupts at start of function  */
        if ( args are incorrect ) {
                restore(mask);      /* restore interrupts before error return   */
                return(SYSERR);
        }
        ... other processing ...
        if ( an error occurs ) {
                restore(mask);      /* restore interrupts before error return   */
                return(SYSERR);
        }
        ... more processing ...
        restore(mask);              /* restore interrupts before normal return */
        return( appropriate value );
}
```

**Figure 6.2** Illustration of the general form of an operating system function.

## 6.7 System Call Return Values SYSERR And OK

We will see that some system calls return a value that relates to the function being performed and others merely return a status to indicate that the call was successful. *Resume* provides an example of the former: it returns the priority of the process that has been resumed. In the case of *resume*, care must be taken to record the priority before calling *ready* because the resumed process may have higher priority than the currently executing process. Thus, as soon as *ready* places the specified process on the ready list and calls *resched*, the new process may begin executing. In fact, an arbitrary delay can occur between the time *resume* calls *ready* and execution continues after the call. During the delay, an arbitrary number of other processes can execute and processes may terminate. Thus, to insure that the returned priority reflects the resumed process's priority at the time of resumption, *resume* makes a copy in local variable *prio* before calling *ready*. *Resume* uses the local copy as the return value.

Xinu defines two constants that are used as return values throughout the system. A function returns *SYSERR* to indicate that an error occurred during processing. That is, a system function returns *SYSERR* if the arguments are incorrect (e.g., outside the acceptable range) or the requested operation could not be completed successfully. A function such as *ready* that does not compute a specific return value uses constant *OK* to indicate that the operation was successful.

## 6.8 Implementation Of Suspend

As the state diagram indicates, *suspend* can only be applied to a process that is current or ready. Suspension of a ready process is trivial: the process must be removed from the ready list and its state must be changed to suspended. No further action is required. Thus, after deleting the process from the ready list and changing the process's state to *PR_SUSP*, *suspend* can restore interrupts and return to its caller. The suspended process will remain ineligible to use the CPU until it has been resumed.

Suspending the current process is almost as easy. Following the steps outlined in Chapter 5, *suspend* must change the state of the current process to *PR-SUSP* and call *resched*. That is, *suspend* sets the state of the current process to the desired next state.

The code for function *suspend* can be found in file *suspend.c*.

```
/* suspend.c - suspend */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  suspend  -  Suspend a process, placing it in hibernation
 *------------------------------------------------------------------------
 */
syscall suspend(
          pid32         pid             /* ID of process to suspend     */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        pri16   prio;                   /* priority to return           */

        mask = disable();
        if (isbadpid(pid) || (pid == NULLPROC)) {
                restore(mask);
                return SYSERR;
        }

        /* Only suspend a process that is current or ready */

        prptr = &proctab[pid];
        if ((prptr->prstate != PR_CURR) && (prptr->prstate != PR_READY)) {
                restore(mask);
                return SYSERR;
        }
        if (prptr->prstate == PR_READY) {
                getitem(pid);               /* remove a ready process   */
                                            /* from the ready list      */
                prptr->prstate = PR_SUSP;
        } else {
                prptr->prstate = PR_SUSP;   /* mark the current process */
                resched();                  /* suspended and reschedule */
        }
        prio = prptr->prprio;
        restore(mask);
        return prio;
}
```

Like *resume*, *suspend* is a system call, which means the function disables interrupts when it is invoked. In addition, *suspend* checks argument *pid* to verify that it is a valid

process ID. Because suspension is only valid for a process that is ready or current, the code verifies that the process is in one of the two valid states. If an error is detected, *suspend* restores interrupts and returns *SYSERR* to the caller.

## 6.9 Suspending The Current Process

The code for suspending the currently executing process raises two interesting points. First, the currently executing process will stop executing, at least temporarily. Thus, before suspending itself, the current process must prearrange for some other process to resume it (or it will remain suspended forever). Second, because it will be suspended, the current process must allow another process to execute. Thus, when suspending the current process, *suspend* must call *resched*. The key idea is that when a process suspends itself, the process remains executing until the call to *resched* selects another process and switches context.

Note that when a process is suspended, *resched* does not place the process on the ready list. In fact, a suspended process is not on a list of suspended processes analogous to the ready list. Ready processes are only kept on an ordered list to speed the search for the highest priority process during rescheduling. Because the system never searches through suspended processes looking for one to resume, the set of suspended processes need not be kept on a list. Thus, before suspending a process, a programmer must arrange a way for the process to be resumed.

## 6.10 Suspend Return Value

*Suspend*, like *resume*, returns the priority of the suspended process to its caller. In the case of a ready process, the value returned will reflect the priority the process had when *suspend* was called. (Once *suspend* disables interrupts, no other process can change priorities in the system, so the priority can be recorded at any time before *suspend* restores interrupts.) In the case of the current process, however, a question arises: should *suspend* return the priority that was in effect when *suspend* was invoked or the priority the process has after the process has been resumed (i.e., when *suspend* returns)? In terms of the code, the question is whether the local copy of the priority should be recorded before the call to *resched* or afterward (the version above records it afterward).

To understand one possible motivation for returning the priority at the time of resumption, consider how the priority can be used to convey information. Suppose, for example, that a process needs to suspend until one of two events occurs. A programmer can assign a unique priority value to each event (e.g., priorities 25 and 26), and arrange the calls to *resume* to set the priority accordingly. The process can then use the priority to determine why it was resumed:

```
newprio = suspend( getpid() );
if (newprio == 25) {
        ... event 1 occurred...
} else {
        ... event 2 occurred...
}
```

## 6.11 Process Termination And Process Exit

Although it freezes a process temporarily, *suspend* saves information about a process so the process can be resumed later. Another system call, *kill*, implements process termination by completely removing a process from the system. Once it has been killed, a process cannot be restarted because *kill* eradicates the entire record and frees the process table entry for reuse by a new process.

The actions taken by *kill* depend on the process state. Before writing the code, a designer needs to consider each possible process state and what it means to terminate a process in that state. We will see, for example, that a process in the ready, sleeping, or waiting states is stored on one of the linked lists in the queue structure, which means *kill* must dequeue the process. In the next chapter, we will see that if a process is waiting for a semaphore, *kill* must adjust the semaphore count. Each of the cases will become clear once we have examined the process state and the functions that control the state. For now, it is sufficient to understand the overall structure of *kill* and see how it handles processes that are current or ready. The code for *kill* appears in file *kill.c* on page 99.

*Kill* checks its argument, *pid*, to ensure that it corresponds to a valid process other than the null process (the null process cannot be killed because it must remain running). *Kill* then decrements *prcount*, a global variable that records the number of active user processes, and calls function *freestk* to free memory that has been allocated for the process's stack. The remaining actions depend on the process's state. For a process that is in the *ready* state, *kill* removes the process from the ready list and then frees the process table entry by assigning value *PR_FREE* to the process's state. Because it no longer appears on the ready list, the process will not be selected for rescheduling; because it has state *PR_FREE*, the entry in the process table can be reused.

Now consider what happens when *kill* needs to terminate the currently executing process. We say that the process *exits*. As before, *kill* validates its argument and decrements the count of active processes. If the current process happens to be the last user process, decrementing *prcount* makes it zero, so *kill* calls function *xdone*, which is explained below. After it marks the current process's state free, *kill* calls *resched* to pass control to another ready process.

```
/* kill.c - kill */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  kill  -  Kill a process and remove it from the system
 *------------------------------------------------------------------------
 */
syscall kill(
          pid32         pid             /* ID of process to kill        */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        int32   i;                      /* index into descriptors       */

        mask = disable();
        if (isbadpid(pid) || (pid == NULLPROC)
            || ((prptr = &proctab[pid])->prstate) == PR_FREE) {
                restore(mask);
                return SYSERR;
        }

        if (--prcount <= 1) {           /* last user process completes  */
                xdone();
        }

        send(prptr->prparent, pid);
        for (i=0; i<3; i++) {
                close(prptr->prdesc[i]);
        }
        freestk(prptr->prstkbase, prptr->prstklen);

        switch (prptr->prstate) {
        case PR_CURR:
                prptr->prstate = PR_FREE;       /* suicide */
                resched();

        case PR_SLEEP:
        case PR_RECTIM:
                unsleep(pid);
                prptr->prstate = PR_FREE;
                break;

        case PR_WAIT:
```

```
                semtab[prptr->prsem].scount++;
                /* fall through */

        case PR_READY:
                getitem(pid);           /* remove from queue */
                /* fall through */

        default:
                prptr->prstate = PR_FREE;
        }

        restore(mask);
        return OK;
}
```

When the last user process exits, *kill* calls *xdone*. In some systems, *xdone* powers down the device. In our example, *xdone* merely prints a message on the console serial line, turns off the LED that indicates the system is running, and halts the processor. The code is found in file *xdone.c*.

```
/* xdone.c - xdone */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  xdone  -  Print system completion message as last thread exits
 *------------------------------------------------------------------------
 */
void    xdone(void)
{
        kprintf("\r\n\r\nAll user processes have completed.\r\n\r\n");
        gpioLEDOff(GPIO_LED_CISCOWHT);  /* turn off LED "run" light     */
        halt();                         /* halt the processor           */
}
```

Why should *kill* invoke function *xdone*? Doing so may seem unnecessary because the code is trivial and could easily be incorporated into *kill* itself. The motivation for using a function stems from a desire to separate functionality: a programmer can change the action taken when all processes exit without modifying *kill*.

A more serious question arises because *xdone* is invoked *before* the last user process has been removed from the system. To understand the problem, consider a fault-tolerant design that restarts processes in the case all processes exit. With the current implementation, one of the process table slots remains in use when *xdone* is called. The exercises consider an alternative implementation.

## 6.12 Process Creation

As we have seen, processes are dynamic — a process can be created at any time. The system call *create* starts a new, independent process. In essence, *create* builds an image of the process as if it had been stopped while running. Once the image has been constructed and the process has been placed on the ready list, *ctxsw* can switch to it.

A look at the code in file *create.c* explains most of the details. *Create* uses function *newpid* to search the process table for a free (i.e., unused) slot. Once a slot has been found, *create* allocates space for the new process's stack, and fills in the process table entry. *Create* calls *getstk* to allocate space for a stack (Chapter 9 discusses memory allocation).

The first argument to *create* specifies the initial function at which the process should start execution. *Create* forms a saved environment on the process's stack as if the specified function had been called. Consequently, we refer to the initial configuration as a *pseudo call*. To build a pseudo call, *create* allocates a context on the process's stack, stores initial values for the registers, and then stores the stack pointer in the corresponding entry of the process table. When *ctxsw* switches to it, the new process begins executing the code for the designated function, obeying the normal calling conventions for accessing arguments and allocating local variables. In short, the initial function for a process behaves exactly as if it had been called.

What value should *create* use as a return address in the pseudo call? The value determines what action the system will take if a process returns from its initial (i.e., top-level) function. Our example system follows a well-known paradigm:

> *If a process returns from the initial (top-level) function in which its execution started, the process exits.*

To be precise, we should distinguish between a return from the function itself and a return from the initial call. To see why, observe that C permits recursive function calls. Thus, if a process begins in function *X* which recursively calls *X*, the first return merely pops one level of recursion and returns to the initial call without causing the process to exit. If the process returns again (or reaches the end of *X*) without making further calls, the process will exit.

To arrange for an exit to occur when the initial call returns, *create* assigns the address of function *userret* as the return address in the pseudo call. The code uses symbolic constant *INITRET*, which has been defined to be function name *userret*.† If during the initial call the process reaches the end of the function or explicitly invokes *return*, control will pass to *userret*. Function *userret* terminates the current process by calling *kill*. File *userret.c*, shown below, contains the code.

*Create* also fills in the process table entry. In particular, *create* makes the state of the newly created process *PR_SUSP*, leaving it suspended, but otherwise ready to run. Finally, *create* returns the process ID of the newly created process; the process must be resumed before it can execute.

†Using a symbolic constant allows the choice to be overridden in a configuration file rather than requiring the code to be changed.

Many of the process initialization details depend on the C run-time environment — one cannot write the code to start a process without understanding the details. For example, *create* arranges for initial arguments to be placed in registers and successive arguments to be placed on the stack. The code that pushes arguments may be difficult to understand because *create* copies the arguments directly from its own run-time stack onto the stack that it has allocated for the new process. To do so, it finds the address of the arguments on its own stack and moves through the list using pointer arithmetic. The details depend on both the processor architecture and the compiler being used.

```
/* create.c - create, newpid */

#include <xinu.h>
#include <mips.h>

static  pid32   newpid(void);

/*------------------------------------------------------------------------
 *  create, newpid  -  Create a process to start running a procedure
 *------------------------------------------------------------------------
 */
pid32   create(
          void          *funcaddr,      /* address of function to run   */
          uint32        ssize,          /* stack size in bytes          */
          pri16         priority,       /* process priority             */
          char          *name,          /* process name (for debugging) */
          uint32        nargs,          /* number of args that follow   */
          ...
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        uint32  *saddr;                 /* stack address                */
        uint32  *savargs;               /* pointer to arg save area     */
        pid32   pid;                    /* ID of newly created process  */
        uint32  *ap;                    /* points to list of var args   */
        int32   pad;                    /* padding needed for arg area  */
        uint32  i;
        void    INITRET(void);

        mask = disable();

        if (   (ssize < MINSTK)
            || (priority <= 0)
            || (((int32)(pid = newpid())) == (int32) SYSERR)
            || ((saddr = (uint32 *)getstk(ssize)) == (uint32 *)SYSERR)) {
```

```
        restore(mask);
        return SYSERR;
}

prcount++;
prptr = &proctab[pid];

/* Initialize process table entry for new process */

prptr->prstate = PR_SUSP;       /* initial state is suspended   */
prptr->prprio = priority;
prptr->prstkptr = (char *)saddr;
prptr->prstkbase = (char *)saddr;
prptr->prstklen = ssize;
prptr->prname[PNMLEN-1] = NULLCH;
for (i=0 ; i<PNMLEN-1 && (prptr->prname[i]=name[i])!=NULLCH; i++)
        ;
prptr->prparent = (pid32)getpid();
prptr->prhasmsg = FALSE;

/* Set up initial device descriptors for the shell              */

prptr->prdesc[0] = CONSOLE;     /* stdin  is CONSOLE device     */
prptr->prdesc[1] = CONSOLE;     /* stdout is CONSOLE device     */
prptr->prdesc[2] = CONSOLE;     /* stderr is CONSOLE device     */

/* Initialize stack as if the process was called                */

*saddr = STACKMAGIC;
*--saddr = pid;
*--saddr = (uint32)prptr->prstklen;
*--saddr = (uint32)prptr->prstkbase - prptr->prstklen
                                + sizeof(int);

if (nargs == 0) {               /* compute padding              */
        pad = 4;
} else if ((nargs%4) == 0) {    /* pad for A0 - A3              */
        pad = 0;
} else {
        pad = 4 - (nargs % 4);
}

for (i = 0; i < pad; i++) {     /* pad stack by inserting zeroes*/
        *--saddr = 0;
}
```

```
        for (i = nargs; i > 0; i--) {   /* reserve space for arguments  */
                *--saddr = 0;
        }
        savargs = saddr;                /* loc. of args on the stack    */

        /* Build the context record that ctxsw expects                  */

        for (i = (CONTEXT_WORDS); i > 0; i--) {
                *--saddr = 0;
        }
        prptr->prstkptr = (char *)saddr;

        /* Address of process entry point                               */

        saddr[(CONTEXT_WORDS) - 1] = (uint32) funcaddr;

        /* Return address value                                         */

        saddr[(CONTEXT_WORDS) - 2] = (uint32) INITRET;

        /* Copy arguments into activation record                        */

        ap = (uint32 *)(&nargs + 1);    /* machine dependent code to    */
        for (i = 0; i < nargs; i++) {   /* copy args onto process stack */
                *savargs++ = *ap++;
        }
        restore(mask);
        return pid;
}

/*------------------------------------------------------------------------
 *  newpid  -  Obtain a new (free) process ID
 *------------------------------------------------------------------------
 */
local   pid32   newpid(void)
{
        uint32  i;                      /* iterate through all processes*/
        static  pid32 nextpid = 1;      /* position in table to try or  */
                                        /*  one beyond end of table     */

        /* Check all NPROC slots */

        for (i = 0; i < NPROC; i++) {
                nextpid %= NPROC;       /* wrap around to beginning */
```

```
                if (proctab[nextpid].prstate == PR_FREE) {
                        return nextpid++;
                } else {
                        nextpid++;
                }
        }
        return (pid32) SYSERR;
}


/* userret.c - userret */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  userret  -  Called when a process returns from the top-level function
 *------------------------------------------------------------------------
 */
void    userret(void)
{
        kill(getpid());                 /* force process exit */
}
```

*Create* introduces an initial transition in the state diagram: a newly created process starts in the *suspended* state. Figure 6.3 illustrates the augmented state diagram.

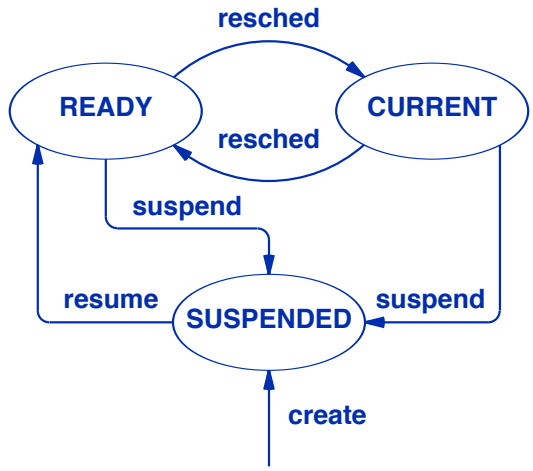



**Figure 6.3** The state diagram showing an initial transition to the *suspended* state.

## 6.13 Other Process Manager Functions

Three additional system calls help manage processes: *getpid*, *getprio*, and *chprio*. As we have seen, *getpid* allows the current process to obtain its process ID, and *getprio* allows a caller to obtain the scheduling priority of an arbitrary process. Another useful system call, *chprio*, allows a process to change the priority of an arbitrary process. The implementation of each of the three functions is straightforward. For example, consider the code for *getprio*. After checking its argument, *getprio* extracts the scheduling priority for the specified process from the process table entry, and returns the priority to the caller.

```
/* getprio.c - getprio */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getprio  -  Return the scheduling priority of a process
 *------------------------------------------------------------------------
 */
syscall getprio(
          pid32         pid             /* process ID                   */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        uint32  prio;                   /* priority to return           */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }
        prio = proctab[pid].prprio;
        restore(mask);
        return prio;
}
```

Because global variable *currpid* contains the ID of the currently executing process, the code for *getpid* is trivial:

```
/* getpid.c - getpid */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getpid  -  Return the ID of the currently executing process
```

```
 *------------------------------------------------------------------------
 */
pid32   getpid(void)
{
        return (currpid);
}
```

Function *chprio* allows the scheduling priority of any process to be changed. The code is found in *chprio.c*.

```
/* chprio.c - chprio */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  chprio  -  Change the scheduling priority of a process
 *------------------------------------------------------------------------
 */
pri16   chprio(
          pid32         pid,            /* ID of process to change      */
          pri16         newprio         /* new priority                 */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        pri16   oldprio;                /* priority to return           */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return (pri16) SYSERR;
        }
        prptr = &proctab[pid];
        oldprio = prptr->prprio;
        prptr->prprio = newprio;
        restore(mask);
        return oldprio;
}
```

The implementation of *chprio* seems to do exactly what is needed. It checks to be sure the specified process exists before changing the priority field in its process table entry. As the exercises point out, however, the code contains two omissions.

## 6.14 Summary

The chapter expands the support for concurrent execution by adding a layer of process management software on top of a scheduler and context switch. The new layer includes routines to *suspend* and *resume* execution as well as routines that *create* a new process or *kill* an existing process. The chapter also examines three additional functions that obtain the ID of the current process (*getpid*), the scheduling priority of the current process (*getprio*), or change the scheduling priority of an arbitrary process (*chprio*). Despite its brevity, the code built thus far forms the basis of a process manager. With proper initialization and support routines, our basic process manager can multiplex a CPU among multiple concurrent computations.

Function *create* forms a new process, and leaves the process in the *suspended* state. *Create* allocates a stack for the new process, and places values on the stack and in the process table such that *ctxsw* can switch to the process and begin execution. The initial values are arranged in a pseudo call, as if the process was called from *userret*. If the process returns from its top-level function, control passes to *userret*, which calls *kill* to terminate the process.

## EXERCISES

**6.1** As the text suggests, a process can tell which of several events triggered resumption if its priority is set to a unique value for each separate call to *resume*. Use the method to create a process that suspends itself and determines which of two other processes resumes it first.

**6.2** Why does *create* build a pseudo-call that returns to *userret* at process exit instead of one that calls *kill* directly?

**6.3** Global variable *prcount* tells the number of active user processes. Carefully consider the code in *kill* and tell whether the count in *prcount* includes the null process?

**6.4** As the text mentions, *kill* calls *xdone* before the last process has been terminated. Change the system so the null process monitors the count of user processes and calls *xdone* when all processes complete.

**6.5** In the previous exercise, what restrictions does the new implementation impose on *xdone* that were not in the current implementation?

**6.6** Some hardware architectures use a special instruction to allow an application program to invoke a system call. Investigate such an architecture, and describe exactly how a system call passes to the correct operating system function.

**6.7** *Create* leaves the new process suspended instead of running. Why?

**6.8** Function *resume* saves the resumed process's priority in a local variable before calling *ready*. Show that if it references *prptr->prprio* after the call to *ready*, *resume* can return a priority value that the resumed process never had (not even after resumption).

**6.9** In function *newpid*, the variable *nextproc* is a global integer that tells the next process table slot to check for a free one. Starting the search from where it left off eliminates looking past the used slots again and again. Speculate on whether the technique is worthwhile in an embedded system.

**6.10** Function *chprio* contains two design flaws. The first arises because the code does not insure that the new priority value is a positive integer. Describe what happens if the priority of a process is set to –1.

**6.11** The second design flaw in *chprio* violates a fundamental design principle. Identify the flaw, describe its consequences, and repair it.

# Chapter Contents

# 7

# Coordination Of Concurrent Processes

*The future belongs to him who knows how to wait.*

— Russian Proverb

## 7.1 Introduction

Previous chapters introduce pieces of a process manager, including scheduling, context switching, and functions that create and terminate processes. This chapter continues the exploration of process management by discussing functions that independent processes use to coordinate and synchronize their actions. The chapter explains the motivation for such primitives and their implementation. The chapter also considers coordination of multiple processors, such as those on a multicore chip.

The next chapter extends our discussion of a process manager by describing a low-level message passing mechanism. Later chapters show how synchronization functions are used to perform I/O.

## 7.2 The Need For Synchronization

Because they execute concurrently, processes need to cooperate when sharing global resources. In particular, an operating system designer must insure that only one process attempts to change a given variable at any time. For example, consider the process table. When a new process is created, a slot in the table must be allocated and values

inserted. If two processes each attempt to create a new process, the system must guarantee that only one of them can execute *create* at a given time, or errors can result.

The previous chapter illustrates one approach system functions can take to guarantee that no other process interferes with them: a function disables interrupts and avoids using any functions that call *resched*. Indeed, system calls such as *suspend*, *resume*, *create*, and *kill* each use the approach.

Why not use the same solution whenever a process needs to guarantee non-interference? The answer is that disabling interrupts has an undesirable global effect on all parts of the system: it stops all activity except for one process, and limits what the process can do. In particular, no I/O can occur while interrupts are disabled. We will learn later that disabling interrupts too long can cause problems (e.g., if packets continue to arrive over a network while interrupts are disabled, the network interface will start to discard them). Therefore, we need a general purpose coordination mechanism that permits arbitrary subsets of the processes to coordinate the use of individual data items without disabling device interrupts for long periods of time, without interfering with processes outside the subset, and without limiting what the running process can do. For example, it should be possible for one process to prohibit changes to a large data structure long enough to format and print the data, without stopping processes that do not need to access the data structure. The mechanism should be transparent: a programmer should be able to understand the consequences of process coordination. Thus, further synchronization mechanisms are needed that:

- Allow a subset of processes to contend for access to a resource
- Provide a policy that guarantees fair access

The first item insures that coordination is local: instead of disabling all interrupts, only those processes contending for a given resource will block waiting for access. Other parts of the system can continue to operate unaffected. The second item insures that if $K$ processes all attempt to access a given resource, each of the $K$ will eventually receive access (i.e., no process is *starved*).

Chapter 2 introduces the fundamental mechanism that solves the problem: *counting semaphores*. The chapter also provides examples that show how processes use semaphores to coordinate. As Chapter 2 indicates, semaphores provide an elegant solution for two problems:

- Mutual exclusion
- Producer–consumer interaction

*Mutual exclusion*. The term *mutual exclusion* is used to describe a situation where a set of processes need to guarantee that only one of them operates at a given time. Mutual exclusion includes access to shared data, but can also include access to an arbitrary shared resource, such as an I/O device.

*Producer–consumer interaction*. We use the term *producer–consumer interaction* to refer to a situation where processes exchange data items. In the simplest form, one process acts as a *producer* by generating a sequence of data items, and another process acts as a *consumer* by accepting the data items. In more complex forms, one or more processes can act as producers and one or more processes can act as consumers. The key to coordination is that each item produced must be received by exactly one consumer (i.e., no items are lost and no items are duplicated).

Both forms of process coordination arise throughout an operating system. For example, consider a set of applications that are producing messages to be displayed on the console. The console device software must coordinate processes to insure that characters do not arrive faster than the hardware can display them. Outgoing characters can be placed in a buffer in memory. Once the buffer fills, the producer must be blocked until space becomes available. Similarly, if the buffer becomes empty, the device stops sending characters. The key idea is that a producer must be blocked when the consumer is not ready to receive data, and a consumer must be blocked when a producer is not ready to send data.

## 7.3 A Conceptual View Of Counting Semaphores

A counting semaphore mechanism that solves both problems described above has a surprisingly elegant implementation. Conceptually, a semaphore, *s*, consists of an integer count and a set of blocked processes. Once a semaphore has been created, processes use two functions, *wait* and *signal*, to operate on the semaphore. A process calls *wait*(*s*) to decrement the count of semaphore *s*, and *signal*(*s*) to increment the count. If the semaphore count becomes negative when a process executes *wait*(*s*), the process is temporarily blocked and placed in the semaphore's set of blocked processes. From the point of view of the process, the call to *wait* does not return for a while. A blocked process becomes ready to run again when another process calls *signal* to increment the semaphore count. That is, if any processes are blocked waiting for a semaphore when *signal* is called, one of the blocked processes will be made ready and allowed to execute. Of course, a programmer must use semaphores with caution: if no process ever signals the semaphore, the blocked processes will wait forever.

## 7.4 Avoidance Of Busy Waiting

What should a process do while waiting on a semaphore? It might seem that after it decrements the semaphore count, a process could repeatedly test the count until the value becomes positive. On a single CPU system, however, such *busy waiting* is unacceptable because other processes will be deprived of the CPU. If no other process receives CPU service, no process can call *signal* to terminate the wait. Therefore, operating systems avoid busy waiting. Instead, semaphore implementations follow an important principle:

*While a process waits on a semaphore, the process does not execute instructions.*

## 7.5 Semaphore Policy And Process Selection

To implement semaphores without busy waiting, an operating system associates a process list with each semaphore. Only the current process can choose to wait on a semaphore. When a process waits on semaphore *s*, the system decrements the count associated with *s*. If the count becomes negative, the process must be blocked. To block a process, the system places the process on the list associated with the semaphore, changes the state so the process is no longer current, and calls *resched* to allow other processes to run.

Later, when *signal* is called on semaphore *s*, the semaphore count is incremented. In addition, the *signal* examines the process list associated with *s*. If the list is not empty (i.e., at least one process is waiting on the semaphore), *signal* extracts a process from the list and moves the process back to the ready list.

A question arises: if multiple processes are waiting, which one should *signal* select? Several policies have been used:

- Highest scheduling priority
- Longest waiting time
- Random
- First-come-first-served

Although it may seem reasonable, selecting the highest priority waiting process violates one of our fundamental principles: fair access. To see why, consider a set of low-priority and high-priority processes that are using a shared resource. Suppose each process uses the resource repeatedly. If the semaphore system always selects a high-priority process and the scheduling policy always gives the CPU to high-priority processes, the low-priority processes can be blocked forever.

Selecting the process that has been waiting longest can lead to a *priority inversion* in the sense that a high-priority process can be blocked while a low-priority process executes. In addition, it can lead to a synchronization problem discussed in an exercise. One way to avoid such problems consists of choosing among waiting processes at *random*. The chief disadvantage of random selection lies in the resources needed. For example, random number generation usually requires multiplication.

Considering the above, many implementations choose a first-come-first-served policy. That is, the system uses a queue to store the set of processes waiting for a given semaphore. When it needs to block a process, *wait* inserts the process at one end of the queue; when it needs to unblock a process, *signal* extracts a process from the other end of the queue. The resulting policy is:

*Semaphore process selection policy: if one or more processes are waiting for semaphore* s *when a* signal *operation occurs for* s*, the process that has been waiting the longest becomes ready.*

## 7.6 The Waiting State

In what state should a process be placed while it is waiting for a semaphore? Because it is neither using the CPU nor eligible to run, the process is neither *current* nor *ready*. The *suspended* state, introduced in the previous chapter cannot be used because functions *suspend* and *resume*, which move processes in and out of the suspended state, have no connection with semaphores. More important, processes waiting for semaphores appear on a list, but suspended processes do not — *kill* must distinguish the two cases when terminating a process. Because existing states do not adequately encompass processes waiting on a semaphore, a new state must be invented. We call the new state *waiting*, and use symbolic constant *PR_WAIT* in the code. Figure 7.1 shows the expanded state transition diagram.

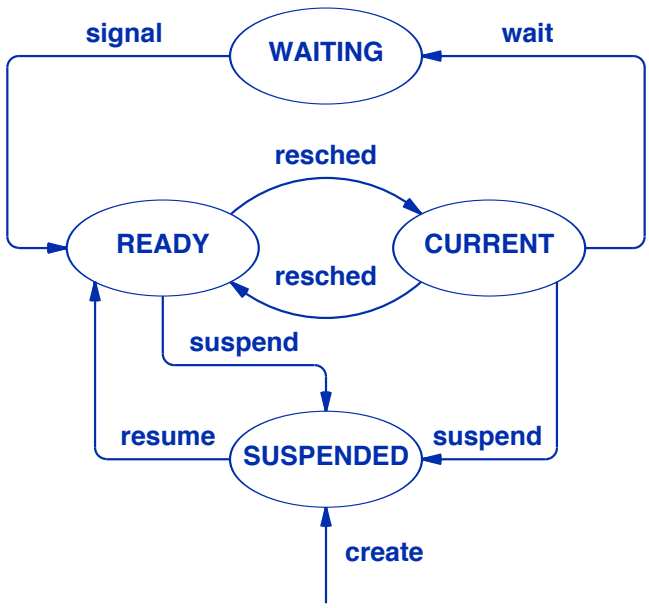



**Figure 7.1** State transitions including the *waiting* state.

## 7.7 Semaphore Data Structures

The example system stores semaphore information in a global semaphore table, *semtab*. Each entry in *semtab* corresponds to one semaphore. The entry contains an integer count and the ID of a queue that can be used to hold waiting processes. The definition of an entry is given by structure *sentry*. File *semaphore.h* contains the details.

```
/* semaphore.h - isbadsem */

#ifndef NSEM
#define NSEM            45      /* number of semaphores, if not defined */
#endif

/* Semaphore state definitions */

#define S_FREE  0               /* semaphore table entry is available   */
#define S_USED  1               /* semaphore table entry is in use      */

/* Semaphore table entry */
struct  sentry  {
        byte    sstate;         /* whether entry is S_FREE or S_USED    */
        int32   scount;         /* count for the semaphore              */
        qid16   squeue;         /* queue of processes that are waiting  */
                                /*     on the semaphore                 */
};

extern  struct  sentry semtab[];

#define isbadsem(s)     ((int32)(s) < 0 || (s) >= NSEM)
```

In structure *sentry*, field *scount* contains the current integer count of the semaphore. The list of processes waiting for a semaphore resides in the queue structure, and field *squeue* gives the index of the head of the list for a given semaphore. The state field, *sstate*, tells whether the entry is currently used (i.e., allocated) or free (currently unallocated).

Throughout the system, semaphores are identified by an integer ID. As with other identification values, semaphore IDs are assigned to make lookup efficient: the semaphore table is an array, and each ID is an index in the array. To summarize:

> *A semaphore is identified by its index in the global semaphore table,* semtab.

## 7.8 The Wait System Call

Recall that the two primary operations on a semaphore are *wait* and *signal*. *Wait* decrements the count of a semaphore. If the count remains nonnegative, *wait* returns to the caller immediately. In essence, a process executing *wait* on a semaphore with a nonpositive count voluntarily gives up control of the CPU. That is, *wait* enqueues the calling process on the list for the semaphore, changes the process state to *PR_WAIT*, and calls *resched* to switch to a ready process. Also recall that our policy maintains the list of processes as a FIFO queue, which means a new process is inserted at the tail of a list. File *wait.c* contains the code.

```
/* wait.c - wait */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  wait  -  Cause current process to wait on a semaphore
 *------------------------------------------------------------------------
 */
syscall wait(
          sid32         sem             /* semaphore on which to wait  */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        struct  sentry *semptr;         /* ptr to sempahore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }

        if (--(semptr->scount) < 0) {           /* if caller must block */
                prptr = &proctab[currpid];
                prptr->prstate = PR_WAIT;       /* set state to waiting */
                prptr->prsem = sem;             /* record semaphore ID  */
                enqueue(currpid,semptr->squeue);/* enqueue on semaphore */
                resched();                      /*   and reschedule     */
        }
```

```
        restore(mask);
        return OK;
}
```

Once enqueued on a semaphore list, a process remains in the waiting state (i.e., not eligible to execute) until the process reaches the head of the queue and some other process signals the semaphore. When the call to *signal* moves a waiting process back to the ready list, the process becomes eligible to use the CPU, and eventually resumes execution. From the point of view of the waiting process, its last act consisted of a call to *ctxsw*. When the process restarts, the call to *ctxsw* returns to *resched*, the call to *resched* returns to *wait*, and the call to *wait* returns to the location from which it was called.

## 7.9 The Signal System Call

Function *signal* takes a semaphore ID as an argument, increments the count of the specified semaphore, and makes the first process ready, if any are waiting. Although it may seem difficult to understand why *signal* makes a process ready even though the semaphore count remains negative or why *wait* does not always enqueue the calling process, the reason is both easy to understand and easy to implement. *Wait* and *signal* maintain the following invariant regarding the count of a semaphore:

> *Semaphore invariant: a nonnegative semaphore count means that the queue is empty; a semaphore count of negative* N *means that the queue contains* N *waiting processes.*

In essence, a count of positive *N* means that *wait* can be called *N* more times before any process blocks. Because *wait* and *signal* each change the semaphore count, they must each adjust the queue length to reestablish the invariant. When it decrements the count, *wait* examines the result, and adds the current process to the queue if the new count is negative. Because it increments the count, *signal* examines the queue and removes a process from the queue if the queue is nonempty.

```
/* signal.c - signal */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  signal  -  Signal a semaphore, releasing a process if one is waiting
 *------------------------------------------------------------------------
 */
```

```
syscall signal(
          sid32         sem             /* id of semaphore to signal    */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  sentry *semptr;         /* ptr to sempahore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }
        semptr= &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }
        if ((semptr->scount++) < 0) {   /* release a waiting process */
                ready(dequeue(semptr->squeue), RESCHED_YES);
        }
        restore(mask);
        return OK;
}
```

## 7.10 Static And Dynamic Semaphore Allocation

An operating system designer must choose between two approaches for semaphore allocation:

- Static allocation: a programmer defines a fixed set of semaphores at compile time; the set does not change as the system runs
- Dynamic allocation: the system includes functions that allow semaphores to be created on demand and deallocated when they are no longer needed

The advantage of static allocation lies in saving space and reducing CPU overhead — the system only contains memory for the needed semaphores, and the system does not require functions to allocate or deallocate semaphores. Thus, the smallest embedded systems use static allocation.

The chief advantage of dynamic allocation arises from the ability to accommodate new uses at run time. For example, a dynamic allocation scheme allows a user to launch an application that allocates a semaphore, terminate the application, and then launch another application. Thus, larger embedded systems and most large operating systems provide dynamic allocation of resources, including semaphores. The next sections show that dynamic allocation does not introduce much additional code.

## 7.11 Example Implementation Of Dynamic Semaphores

Xinu provides a limited form of dynamic allocation: processes can create semaphores dynamically, and a given process can create multiple semaphores, provided the total number of semaphores allocated simultaneously does not exceed a predefined maximum. Furthermore, to minimize the allocation overhead, the system preallocates a list in the queue structure for each semaphore when the operating system boots. Thus, only a small amount of work needs be done when a process creates a semaphore.

Two system calls, *semcreate* and *semdelete*, handle dynamic semaphore allocation and deallocation. *Semcreate* takes an initial semaphore count as an argument, allocates a semaphore, assigns the semaphore the specified count, and returns the semaphore ID. To preserve the semaphore invariant, the initial count must be nonnegative. Therefore, *semcreate* begins by testing its argument. If the argument is valid, *semcreate* searches the semaphore table, *semtab*, for an unused entry and initializes the count. To search the table, *semcreate* calls procedure *newsem*, which iterates through all *NSEM* entries of the table. If no free entry is found, *newsem* returns *SYSERR*. Otherwise, *newsem* changes the state of the entry to *S_USED*, and returns the table index.

Once a table entry has been allocated, *semcreate* only needs to initialize the count and return the index of the semaphore to its caller; the head and tail of a queue used to store waiting processes have been allocated when the operating system boots. File *semcreate.c* contains the code for function *newsem* as well as function *semcreate*. Note the use of a static index variable *nextsem* to optimize searching (i.e., allow a search to start where the last search left off).

```
/* semcreate.c - semcreate, newsem */

#include <xinu.h>

local   sid32   newsem(void);

/*------------------------------------------------------------------------
 *  semcreate  -  create a new semaphore and return the ID to the caller
 *------------------------------------------------------------------------
 */
sid32   semcreate(
          int32         count           /* initial semaphore count      */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        sid32   sem;                    /* semaphore ID to return       */

        mask = disable();

        if (count < 0 || ((sem=newsem())==SYSERR)) {
```

```
                restore(mask);
                return SYSERR;
        }
        semtab[sem].scount = count;     /* initialize table entry       */

        restore(mask);
        return sem;
}

/*------------------------------------------------------------------------
 *  newsem  -  allocate an unused semaphore and return its index
 *------------------------------------------------------------------------
 */
local   sid32   newsem(void)
{
        static  sid32   nextsem = 0;    /* next semaphore index to try  */
        sid32   sem;                    /* semaphore ID to return       */
        int32   i;                      /* iterate through # entries    */

        for (i=0 ; i<NSEM ; i++) {
                sem = nextsem++;
                if (nextsem >= NSEM)
                        nextsem = 0;
                if (semtab[sem].sstate == S_FREE) {
                        semtab[sem].sstate = S_USED;
                        return sem;
                }
        }
        return SYSERR;
}
```

## 7.12 Semaphore Deletion

Function *semdelete* reverses the actions of *semcreate*. *Semdelete* takes the index of a semaphore as an argument and releases the semaphore table entry for subsequent use. Deallocating a semaphore requires three steps. First, *semdelete* verifies that the argument specifies a valid semaphore ID and that the corresponding entry in the semaphore table is currently in use. Second, *semdelete* sets the state of the entry to *S_FREE* to indicate that the table entry can be reused. Finally, *semdelete* iterates through the set of processes that are waiting on the semaphore and makes each process ready. File *semdelete.c* contains the code.

```
/* semdelete.c - semdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
 * semdelete  --  Delete a semaphore by releasing its table entry
 *------------------------------------------------------------------------
 */
syscall semdelete(
          sid32         sem             /* ID of semaphore to delete    */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  sentry *semptr;         /* ptr to semaphore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }
        semptr->sstate = S_FREE;

        while (semptr->scount++ < 0) {  /* free all waiting processes   */
                ready(getfirst(semptr->squeue), RESCHED_NO);
        }
        resched();
        restore(mask);
        return OK;
}
```

If processes remain enqueued on a semaphore when the semaphore is deallocated, an operating system must handle each of the processes. In the example implementation, *semdelete* places each waiting process back on the ready list, allowing the process to resume execution as if the semaphore had been signaled. The example only represents one strategy, and other strategies are possible. For example, some operating systems consider it an error to attempt to deallocate a semaphore on which processes are waiting. The exercises suggest exploring alternatives.

Note that the code to make processes ready does not reschedule after a process has been placed on the ready list. Instead, each call to *ready* specifies *RESCHED_NO*. After all waiting processes have been moved to the ready list, the code calls *resched* explicitly to reestablish the scheduling invariant.

## 7.13 Semaphore Reset

It is sometimes convenient to reset the count of a semaphore without incurring the overhead of deleting an old semaphore and acquiring a new one. The system call *semreset*, shown in file *semreset.c* below, resets the count of a semaphore.

```
/* semreset.c - semreset */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  semreset  --  reset a semaphore's count and release waiting processes
 *------------------------------------------------------------------------
 */
syscall semreset(
          sid32         sem,            /* ID of semaphore to reset     */
          int32         count           /* new count (must be >= 0)     */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  sentry *semptr;         /* ptr to semaphore table entry */
        qid16   semqueue;               /* semaphore's process queue ID */
        pid32   pid;                    /* ID of a waiting process      */

        mask = disable();

        if (count < 0 || isbadsem(sem) || semtab[sem].sstate==S_FREE) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        semqueue = semptr->squeue;      /* free any waiting processes */
        while ((pid=getfirst(semqueue)) != EMPTY)
                ready(pid,RESCHED_NO);
        semptr->scount = count;         /* reset count as specified */
        resched();
        restore(mask);
```

```
        return(OK);
}
```

*Semreset* must preserve the semaphore invariant. Rather than build a general-purpose solution that allows a caller to specify an arbitrary semaphore count, our implementation takes a simplified approach by requiring the new count to be non-negative. As a result, once the semaphore count has been changed, the queue of waiting processes will be empty. As with *semdelete*, *semreset* must be sure that no processes are already waiting on the semaphore. Thus, after checking its arguments and verifying that the semaphore exists, *semreset* iterates through the list of waiting processes, removing each from the semaphore queue and making the process ready to execute. Once all waiting processes have been removed, *semreset* assigns the new value to the semaphore count, reschedules (in case a waiting process had higher priority), and returns to its caller.

## 7.14 Coordination Across Parallel Processors (Multicore)

The semaphore system described above works well on a computer that has a single CPU. However, many modern processor chips include multiple cores. One core is usually dedicated to run operating system functions, and other cores are used to execute user applications. On such systems, using semaphores supplied by the operating system to coordinate processes can be inefficient. To see why, consider what happens when an application running on core 2 needs exclusive access to a specific memory location. The application process calls *wait*, which must pass the request to the operating system on core 1. Core 2 must interrupt core 1 to make the request. Furthermore, while it runs an operating system function, core 1 disables interrupts, which defeats one of the reasons to use semaphores.

Some multiprocessor systems supply hardware primitives, known as *spin locks*, that allow multiple processors to contend for mutually exclusive access. The hardware defines a set of $K$ spin locks ($K$ might be less than 1024). Conceptually, each of the spin locks is a single bit, and the spin locks are initialized to zero. Each processor includes a special instruction called a *test-and-set* that performs two operations atomically: it sets a spin lock to 1 and returns the value of the spin lock before the operation. The hardware guarantee of atomicity means that if two or more processors attempt to set a given spin lock simultaneously, one of them will receive 0 as the previous value and the others will receive 1. Once it finishes, the processor that obtained the lock resets the value to 0, allowing another processor to obtain the lock.

To see how spin locks work, suppose two processors need exclusive access to a shared data item and are using spin lock 5. When a processor wants to obtain mutually exclusive access, the processor executes a loop:†

†Because it uses hardware instructions, test-and-set code is usually written in assembly language; it is shown in pseudo code for clarity.

```
while (test_and_set(5)) {
        ;
}
```

The loop repeatedly uses the test-and-set instruction to set spin lock 5. If the lock was set before the instruction executed, the instruction will return 1, and the loop will continue. If the lock was not set before the instruction executed, the hardware will return 0, and the loop terminates. If multiple processors are all trying to set spin lock 5 at the same time, the hardware guarantees that only one will be granted access. Thus, test_and_set is analogous to *wait*.

Once a processor finishes using the shared data, the processor executes an instruction that clears the spin lock:

```
clear(5);
```

On multiprocessor machines that do not have spin lock hardware, vendors include instructions that can be used to create a spin lock. For example, Intel multicore processors rely on an atomic *compare-and-swap* instruction in memory. If multiple cores attempt to execute the instruction at the same time, one of them will succeed and the others will all find that the comparison fails. A programmer can use such instructions to build the equivalent of a spin lock.

It may seem that a spin lock is wasteful because a processor merely blocks in a loop until access is granted. However, if the probability of two processors contending for a spin lock at the same time is low, the mechanism is much more efficient than a system call. Therefore, a programmer must be careful in choosing when to use spin locks and when to use system calls.

## 7.15 Perspective

The counting semaphore abstraction is significant for two reasons. First, it provides a powerful mechanism that can be used to control both mutual exclusion and producer–consumer synchronization, the two primary process coordination paradigms. Second, the implementation is surprisingly compact and extremely efficient. To appreciate the small size, reconsider functions *wait* and *signal*. If the code to test arguments and returns results is removed, only a few lines of code remain. As we examine the implementation of other abstractions, the point will become more significant: despite their importance, only a trivial amount of code is needed to implement counting semaphores.

## 7.16 Summary

Instead of disabling interrupts, which stops all activities other than the current process, operating systems offer synchronization primitives that allow subsets of processes to coordinate without affecting other processes. A fundamental coordination mechanism, known as a counting semaphore, allows processes to coordinate without using

busy-waiting. Each semaphore consists of an integer count plus a queue of processes. The semaphore adheres to an invariant that specifies a count of negative *N* means the queue contains *N* processes.

The two fundamental primitives, *signal* and *wait*, permit a caller to increment or decrement the semaphore count. If a call to *wait* makes the semaphore count negative, the calling process is placed in the *waiting* state and the CPU passes to another process. In essence, a process that waits for a semaphore voluntarily enqueues itself on the list of processes waiting for the semaphore, and calls *resched* to allow other processes to execute.

Either static or dynamic allocation can be used with semaphores. The example code includes functions *semcreate* and *semdelete* to permit dynamic allocation. If a semaphore is deallocated while processes are waiting, the processes must be handled. The example code makes the processes ready as if the semaphore had been signaled.

Multiprocessors can use a mutual exclusion mechanism known as spin locks. Although spin locks seem inefficient because they require a processor to repeatedly test for access, they can be more efficient than an arrangement where one processor interrupts another to place a system call.

## EXERCISES

**7.1** The text notes that some operating systems consider semaphore deletion to be in error if processes remain enqueued waiting for the semaphore. Rewrite *semdelete* to return *SYSERR* for a busy semaphore.

**7.2** As an alternative to the semaphore deletion mechanism illustrated in the chapter, consider deferral. That is, rewrite *semdelete* to place a deleted semaphore in a *deferred* state until all processes have been signaled. Modify *signal* to release the semaphore table entry when the last waiting process has been removed from the queue.

**7.3** In the previous exercise, can deferred deletion have unexpected side effects? Explain.

**7.4** As a further alternative to the deferred deletion of an active semaphore, modify *wait* to return a value *DELETED* if the semaphore was deleted while the calling process was waiting. (Choose a value for *DELETED* that differs from *SYSERR* and *OK*.) How can a process determine whether the semaphore on which it was waiting has been deleted? Be careful: remember that a high priority process can execute at any time. Thus, after a low-priority process becomes ready, a higher priority process can obtain the CPU and create a new semaphore that reuses the semaphore table entry before the low-priority process completes execution of *wait*. Hint: consider adding a sequence field to the semaphore table entry.

**7.5** Instead of allocating a central semaphore table, arrange to have each process allocate space for semaphore entries as needed, and use the address of an entry as the semaphore ID. Compare the approach to the centralized table in the example code. What are the advantages and disadvantages of each?

**7.6** *Wait*, *signal*, *semcreate*, and *semdelete* coordinate among themselves for use of the semaphore table. Is it possible to use a semaphore to protect use of the semaphore table? Explain.

**7.7** Why does *semdelete* call *ready* without rescheduling?

**7.8** Consider a possible optimization: arrange for *semdelete* to examine the priority of each waiting process before the process is placed on the ready list. If none of the processes has higher priority than the current process, do not call *resched*. What is the cost of the optimization and what is the potential savings?

**7.9** Construct a new system call, *signaln*(*sem*, *n*) that signals semaphore *sem n* times. Can you find an implementation that is more efficient than *n* calls to *signal*? Explain.

**7.10** The example code uses a FIFO policy for semaphores. That is, when a semaphore is signaled, the process that has been waiting the longest becomes ready. Imagine a modification in which the processes waiting for a semaphore are kept on a priority queue ordered by process priority (i.e., when a semaphore is signaled, the highest priority waiting process becomes ready). What is the chief disadvantage of a priority approach?

**7.11** Languages meant specifically for writing concurrent programs often have coordination and synchronization embedded in the language constructs directly. For example, it might be possible to declare procedures in groups such that the compiler automatically inserts code to prohibit more than one process from executing a given group. Find an example of a language designed for concurrent programming, and compare process coordination with the semaphores in the Xinu code. When a programmer is required to manipulate semaphores explicitly, what types of mistakes can a programmer make?

**7.12** When it moves a waiting process to the ready state, *wait* sets field *prsem* in the process table entry to the ID of the semaphore on which the process is waiting. Will the value ever be used?

**7.13** If a programmer makes a mistake, it is more likely that the error will produce 0 or 1 than an arbitrary integer. To help prevent errors, change *newsem* to begin allocating semaphores from the high end of the table, leaving slots 0 and 1 unused until all other entries have been exhausted. Suggest better ways of identifying semaphores that increase the ability to detect errors.

**7.14** Function *semdelete* behaves in an unexpected way when deleting a semaphore with a non-negative count. Identify the behavior and rewrite the code to correct it.

**7.15** Draw a call graph of all operating system functions from Chapters 4 through 7, showing which functions a given function invokes. Can a multi-level structure be deduced from the graph? Explain.

# Chapter Contents

# 8

# Message Passing

*The message of history is clear: the past lies before us.*

— Anonymous

## 8.1 Introduction

Previous chapters explain the basic components of a process manager, including: scheduling, context switching, and counting semaphores that provide coordination among concurrent processes. The chapters show how processes are created and how they terminate, and explain how an operating system keeps information about each process in a central table.

This chapter concludes our examination of basic process management facilities. The chapter introduces the concept of message passing, describes possible approaches, and shows an example of a low-level message passing system. Chapter 11 explains how a high-level message passing facility can be built using the basic process management mechanisms.

## 8.2 Two Types Of Message Passing Services

We use the term *message passing* to refer to a form of inter-process communication in which one process transfers (usually a short amount of) data to another. In some systems, processes deposit and retrieve messages from named *pickup points* that are sometimes called *mailboxes*. In other systems, a message must be addressed directly to a process. Message passing is both convenient and powerful, and some operating systems use it as the basis for all communication and coordination among processes. For

example, an operation such as transmitting data across a computer network can be implemented using message passing primitives.

Some message passing facilities provide process coordination because the mechanism delays a receiver until a message arrives. Thus, message passing can replace process suspension and resumption. Can message passing also replace synchronization primitives such as semaphores? The answer depends on the implementation of message passing. There are two types:

- *Synchronous*. If a receiver attempts to receive a message before the message has arrived, the receiver blocks; if a sender tries to send a message before a receiver is ready, the sender blocks. Sending and receiving processes must coordinate or one can become blocked waiting for the other.
- *Asynchronous*. A message can arrive at any time, and a receiver is notified. A receiver does not need to know in advance how many messages will arrive or how many senders will send messages.

Although it may lack generality and convenience, a synchronous message passing facility can serve in place of a semaphore mechanism. For example, consider a producer–consumer paradigm. Each time it generates new data, a producer process can send a message to the consumer process. Similarly, instead of waiting on a semaphore, the consumer can wait for a message. Using message passing to implement mutual exclusion is more complex, but usually possible.

The chief advantage of a synchronous message passing system arises because it fits well with a traditional computational paradigm. To receive a message in a synchronous system, a process calls a system function, and the call does not return until a message arrives. In contrast, an asynchronous message passing system either requires a process to *poll* (i.e., check for a message periodically) or requires a mechanism that allows the operating system to stop a process temporarily, allow the process to handle a message, and then resume normal execution. Although it introduces additional overhead or complexity, asynchronous message passing can be convenient if a process does not know how many messages it will receive, when the messages will be sent, or which processes will send messages.

## 8.3 Limits On Resources Used By Messages

Xinu supports two forms of message passing that illustrate a completely synchronous paradigm and a partially asynchronous paradigm. The two facilities also illustrate the difference between direct and indirect message delivery: one provides a direct exchange of messages among processes, and the other arranges for messages to be exchanged through rendezvous points. This chapter begins the discussion by examining a facility that provides direct communication from one process to another. Chapter 11 discusses a second message passing facility. Separating message passing into two in-

dependent pieces has the advantage of making low-level message passing among processes efficient, while allowing a programmer to choose a more complex rendezvous approach when needed.

The Xinu process-to-process message passing system has been designed carefully to ensure that a process does not block while sending a message, and waiting messages do not consume all of memory. To make such guarantees, the message passing facility follows three guidelines:

- *Limited message size*. The system limits each message to a small, fixed size. In our example code, each message consists of a single word (i.e., an integer or a pointer).
- *No message queues*. The system permits a given process to store only one unreceived message per process at any time. There are no message queues.
- *First message semantics*. If several messages are sent to a given process before the process receives any of them, only the first message is stored and delivered; subsequent senders do not block.

The concept of *first message semantics* makes the mechanism useful for determining which of several events completes first. A process that needs to wait for events can arrange for each event to send a unique message. The process then waits for a message, and the operating system guarantees that the process will receive the first message that is sent.

## 8.4 Message Passing Functions And State Transitions

Three system calls manipulate messages: *send*, *receive*, and *recvclr*. *Send* takes a message and a process ID as arguments, and delivers the message to the specified process. *Receive*, which does not require arguments, causes the current process to wait until a message arrives, and then returns the message to its caller. *Recvclr* provides a non-blocking version of *receive*. If the current process has received a message when *recvclr* is called, the call returns the message exactly like *receive*. If no message is waiting, however, *recvclr* returns the value *OK* to its caller immediately, without delaying to wait for a message to arrive. As the name implies, *recvclr* can be used to remove an old message before engaging in a round of message passing.

The question arises: in what state should a process be while waiting for a message? Because waiting for a message differs from being ready to execute, waiting for a semaphore, waiting for the CPU, suspended animation, or current execution, none of the existing states suffices. Thus, another state must be added to our design. The new state, *receiving*, is denoted in the example software with the symbolic constant *PR_RECV*. Adding the state produces the transition diagram illustrated in Figure 8.1.

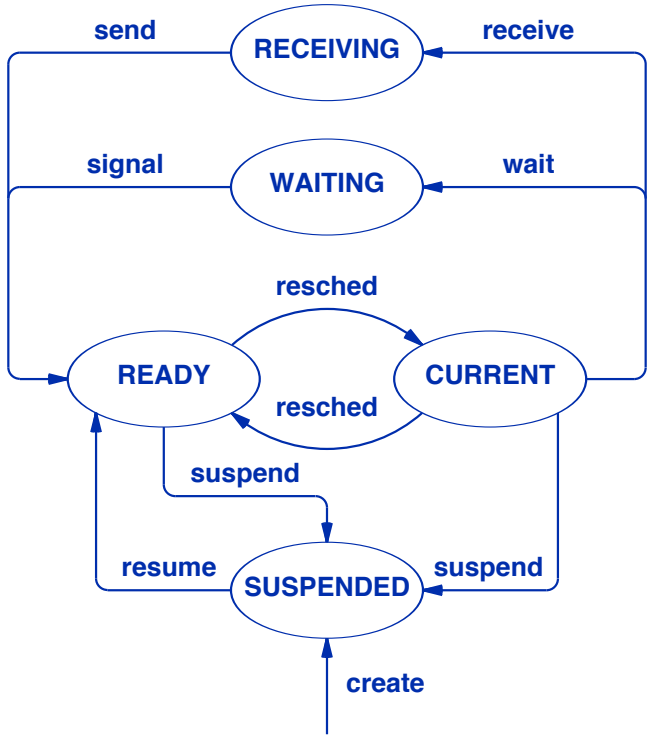



**Figure 8.1** Illustration of state transitions including the *receiving* state.

## 8.5 Implementation Of Send

A message passing system requires agreement between senders and receivers because a sender must store a message in a location from which the receiver can extract the message. A message cannot be stored in the sender's memory because a sending process can exit before the message is received. Most operating systems do not permit a sender to place a message in a receiver's address space because allowing a process to write into the memory allocated to another process poses a security threat. In our example system, restrictions on the size of messages eliminate the problem. Our implementation reserves space for one message in field *prmsg* of the recipient's process table entry.

To deposit a message, function *send* first checks that the specified recipient process exists. It then checks to insure the recipient does not have a message outstanding. To do so, *send* examines field *prhasmsg* in the recipient's process table entry. If the recipient has no outstanding message, *send* deposits the new message in the *prmsg* field and sets *prhasmsg* to *TRUE* to indicate that a message is waiting. As a final step, if the recipient is waiting for the arrival of a message (i.e., the recipient process has state *PR_RECV* or state *PR_RECTIM*), *send* calls *ready* with argument RESCHED_YES to

make the process ready and re-establish the scheduling invariant. In the case of *PR_RECTIM*, which is discussed later in the text, *send* must first call *unsleep* to remove the process from the queue of sleeping processes. File *send.c* contains the code.

```
/* send.c - send */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  send  -  pass a message to a process and start recipient if waiting
 *------------------------------------------------------------------------
 */
syscall send(
          pid32         pid,            /* ID of recipient process      */
          umsg32        msg             /* contents of message          */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }

        prptr = &proctab[pid];
        if ((prptr->prstate == PR_FREE) || prptr->prhasmsg) {
                restore(mask);
                return SYSERR;
        }
        prptr->prmsg = msg;             /* deliver message              */
        prptr->prhasmsg = TRUE;         /* indicate message is waiting  */

        /* If recipient waiting or in timed-wait make it ready */

        if (prptr->prstate == PR_RECV) {
                ready(pid, RESCHED_YES);
        } else if (prptr->prstate == PR_RECTIM) {
                unsleep(pid);
                ready(pid, RESCHED_YES);
        }
        restore(mask);          /* restore interrupts */
        return OK;
}
```

## 8.6 Implementation Of Receive

A process, calls *receive* (or *recvclr*) to obtain an incoming message. *Receive* examines the process table entry for the current process, and uses the *prhasmsg* field to determine whether a message is waiting. If no message has arrived, *receive* changes the process state to *PR_RECV*, and calls *resched*, to allow other processes to run. When another process sends the receiving process a message, the call to *resched* returns. Once execution passes the *if* statement, *receive* extracts the message, sets *prhasmsg* to *FALSE*, and returns the message to its caller. File *receive.c* contains the code:

```
/* receive.c - receive */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  receive  -  wait for a message and return the message to the caller
 *------------------------------------------------------------------------
 */
umsg32  receive(void)
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        umsg32  msg;                    /* message to return            */

        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == FALSE) {
                prptr->prstate = PR_RECV;
                resched();              /* block until message arrives  */
        }
        msg = prptr->prmsg;             /* retrieve message             */
        prptr->prhasmsg = FALSE;        /* reset message flag           */
        restore(mask);
        return msg;
}
```

Look carefully at the code and notice that *receive* copies the message from the process table entry into local variable *msg* and then returns the value in *msg*. Interestingly, *receive* does not modify field *prmsg* in the process table. Thus, it may seem that a more efficient implementation would avoid copying into a local variable and simply return the message from the process table:

```
return proctab[currpid].prmsg;
```

Unfortunately, such an implementation is incorrect; an exercise asks readers to consider why the implementation can produce incorrect results.

## 8.7 Implementation Of Non-Blocking Message Reception

*Recvclr* operates much like *receive* except that it always returns immediately. If a message is waiting, *recvclr* returns the message; otherwise, *recvclr* returns *OK*.

```
/* recvclr.c - recvclr */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  recvclr  -  clear incoming message, and return message if one waiting
 *------------------------------------------------------------------------
 */
umsg32  recvclr(void)
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        umsg32  msg;                    /* message to return            */

        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == TRUE) {
                msg = prptr->prmsg;     /* retrieve message             */
                prptr->prhasmsg = FALSE;/* reset message flag           */
        } else {
                msg = OK;
        }
        restore(mask);
        return msg;
}
```

## 8.8 Perspective

Like the counting semaphore abstraction in the previous chapter, the code for a basic message passing facility is extremely compact and efficient. Look at the functions and notice how few lines of code perform each operation. Furthermore, observe that storing the message buffer in the process table is important because doing so isolates message passing from memory management and allows message passing to be positioned at a low level in the hierarchy.

## 8.9 Summary

Message passing facilities provide inter-process communication that allows a process to send information to another process. A completely synchronous message passing system blocks either the sender or receiver, depending on how many messages have been sent and received. Our example system includes two facilities for message passing: a low-level mechanism that provides direct communication among processes, and a high-level mechanism that uses rendezvous points.

The Xinu low-level message passing mechanism limits the message size to a single word, restricts each process to at most one outstanding message, and uses first-message semantics. Message storage is associated with the process table — a message sent to process *P* is stored in the process table entry for *P*. The use of first-message semantics allows a process to determine which of several events occurs first.

The low-level message facility comprises three functions: *send*, *receive*, and *recvclr*. Of the three functions, only *receive* is blocking — it blocks the calling process until a message arrives. A process can use *recvclr* to remove an old message before starting an interaction that uses message passing.

## EXERCISES

**8.1** Write a program that prints a prompt, and then loops printing the prompt again every 8 seconds until someone types a character. (Hint: sleep(8) delays the calling process for 8 seconds.)

**8.2** Assume *send* and *receive* do not exist, and build code to perform message passing using *suspend* and *resume*.

**8.3** The example implementation uses first-message semantics. What facilities exist to handle last-message semantics?

**8.4** Implement versions of *send* and *receive* that record up to *K* messages per process (make successive calls to *send* block).

**8.5** Investigate systems in which the innermost level of the system implements message passing instead of context switching. What is the advantage? The chief liability?

**8.6** Consider the modification of *receive* mentioned in the text that returns the message directly from the process table entry:

```
return proctab[currpid].prmsg;
```

Explain why such an implementation is incorrect.

**8.7** Implement a version of *send* and *receive* that define a fixed set of thirty-two possible messages. Instead of using integers to represent messages, use one bit of a word to represent each message, and allow a process to accumulate all thirty-two messages.

**8.8** Observe that because *receive* uses *SYSERR* to indicate an error, sending a message with the same value as *SYSERR* is ambiguous. Furthermore, *recvclr* returns *OK* if no message is waiting. Modify *recvclr* to return *SYSERR* if no message is waiting, and modify *send* so it refuses to send *SYSERR* (i.e., checks its argument and returns an error if the value is *SYSERR*).

# NOTES

# Chapter Contents

# 9

# Basic Memory Management

*Memory is the ghost of experience.*

— Anonymous

## 9.1 Introduction

Previous chapters explain concurrent computation and the facilities that an operating system provides to manage concurrent processes. The chapters discuss process creation and termination, scheduling, context switching, coordination, and inter-process communication.

This chapter begins the discussion of a second key topic: facilities that an operating system uses to manage memory. The chapter focuses on basics: dynamic allocation of stack and heap storage. It presents a set of functions that allocate and free memory, and explains how an embedded system handles memory for processes. The next chapter continues the discussion of memory management by describing address spaces, high-level memory management facilities, and virtual memory.

## 9.2 Types Of Memory

Because it is essential for program execution and data storage, main memory ranks high among the important resources that an operating system manages. An operating system maintains information about the size and location of available free memory blocks, and allocates memory to concurrent programs upon request. The system recovers the memory allocated to a process when the process terminates, making the memory available for reuse.

In most embedded systems, main memory consists of a contiguous set of locations with addresses *0* through *N – 1*; the code and data remain resident in memory. A small embedded system can use two memory technologies:

- *Read-Only Memory* (*ROM*), which may include Flash, used to store program code and constants
- *Random Access Memory* (*RAM*) used to hold variables that can change during execution

In terms of the addresses used, the two types of memory occupy separate locations. For example, addresses *0* through *K – 1* might correspond to ROM, and addresses *K* through *N – 1* might correspond to RAM.†

Some systems further distinguish regions of memory by using specific types of memory technologies. For example, RAM can be divided into two regions that use:

- *Static RAM* (*SRAM*): faster, but more expensive
- *Dynamic RAM* (*DRAM*): less expensive, but slower

Because SRAM is more expensive, systems usually have a small amount of SRAM and a larger amount of DRAM. If memory types differ, a programmer must carefully place variables and code that are referenced the most frequently in SRAM, and items that are referenced less often in DRAM.

## 9.3 Definition Of A Heavyweight Process

Large operating systems provide protection mechanisms that prevent an application from reading or modifying areas of memory that have been assigned to another application. In particular, Chapter 10 discusses how a separate virtual address space can be assigned to each computation. The approach, which is known as a *heavyweight process abstraction*, creates an address space, and then creates a process to run in the address space. Usually, code for a heavyweight process is loaded dynamically — an application must be compiled and stored in a file on disk before the application can be used in a heavyweight process. Thus, when creating the heavyweight process, a programmer specifies the file on disk that contains compiled code, and the operating system loads the specified application into a new virtual address space and starts a process executing the application.

Interestingly, some operating systems that support a heavyweight process abstraction use a hybrid approach that includes a *lightweight process abstraction* (i.e., *threads of execution*). Instead of a single process executing in an address space, the operating system permits a user to create multiple threads that each execute in the address space.

In a hybrid system, a thread is similar to a process in Xinu. Each thread has a separate run-time stack used to hold local variables and function calls; the stacks are allocated from the data area in the heavyweight process's address space. All the threads

†Usually, *K* and *N* are each a power of 2.

within a given heavyweight process share global variables; the global variables are allocated in the data area of the heavyweight process's address space. Sharing implies the need for coordination — threads within a heavyweight process must use synchronization primitives, such as semaphores, to control access to the variables they share. Figure 9.1 illustrates a hybrid system.

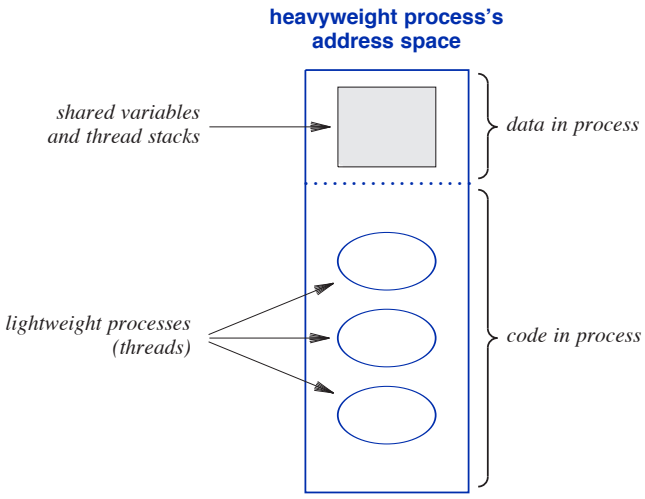



**Figure 9.1** Illustration of a heavyweight process abstraction concept with multiple lightweight processes (threads) sharing the address space.

## 9.4 Memory Management In A Small Embedded System

The largest embedded systems, such as those used in video game consoles, have secondary storage and the memory management hardware needed to support virtual address spaces. On the smallest embedded systems, however, the hardware cannot support multiple address spaces, nor can it protect processes from one another. As a consequence, the operating system and all processes occupy a single address space.

Although running multiple processes in a single address space does not offer protection, the approach has advantages. Because they can pass pointers among themselves, processes can share large amounts of data without copying from one address space to another. Furthermore, the operating system itself can easily dereference an arbitrary pointer because the interpretation of an address does not depend on the process context. Finally, having just one address space makes the memory manager much simpler than the memory managers found in more sophisticated systems.

## 9.5 Program Segments And Regions Of Memory

When Xinu is compiled for a small embedded system, the memory image is divided into four contiguous regions:

- Text segment
- Data segment
- Bss segment
- Free space

*Text segment*. The text segment, which begins at memory location zero, contains compiled code for each of the functions in the memory image. The text segment may also contain constants (e.g., string constants). If the hardware includes a protection mechanism, addresses in the text segment are classified as *read-only*, which means that an error occurs if the program attempts to store into any of the text locations at run-time.

*Data segment*. The data segment, which follows the text segment, contains storage for all global variables that are assigned an initial value. Values in the data segment may be accessed or modified (i.e., read or written).

*Bss segment*. The term *bss* abbreviates *block started by symbol*, and is taken from the assembly language of a PDP-11, the computer on which C was designed. The bss segment, which follows the data segment, contains global variables that are not initialized explicitly. Following C conventions, Xinu writes zero into each bss location before execution begins.

*Free space*. Memory beyond the bss segment is considered *free* (i.e., unallocated) when execution begins. The next sections describe how the operating system uses the free space.

As described in Chapter 3, a C program loader defines three external symbols, *etext*, *edata*, and *end*,† that correspond to the first memory location beyond the text segment, the first memory location beyond the data segment, and the first memory location beyond the bss segment. Figure 9.2 illustrates the memory layout when Xinu begins to execute and the three external symbols.

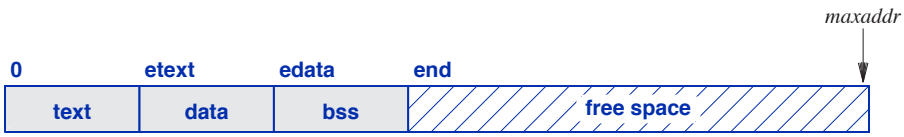



**Figure 9.2** Illustration of the memory layout when Xinu begins.

The external symbols shown in Figure 9.2 are not variables, but instead are names assigned to memory locations when the image is linked. Thus, a program should only

†External symbol names have an underscore prepended by the loader. Thus, *etext* becomes *_etext*.

use one of the external symbols to reference a location and should not load a value. For example, the text segment occupies memory locations *0* through *etext–1*. To compute the size, a program declares *etext* to be an external integer, and references *&etext* in an expression.

How can a program determine the size of the free space? The operating system must either be configured with the highest physical memory address or must probe addresses at startup.† At startup, our system places the highest valid memory address in global variable *maxheap*. Thus, to compute the initial size of the free space, the memory management code can take the difference between the address in variable *maxheap* and the value *&end*.

## 9.6 Dynamic Memory Allocation In An Embedded System

Although they are allocated fixed locations in the address space and must remain resident in physical memory at all times, program text and global variables only account for part of the memory used by an executing process. The other two types of memory are:

- Stack
- Heap

*Stack*. Each process needs space for a *stack* that holds the activation record associated with each function the process invokes. In addition to arguments, an activation record contains storage for local variables.

*Heap*. A process or set of processes may also use *heap* storage to hold dynamically allocated variables that persist independent of specific function calls.

Xinu accommodates both types of dynamic memory. First, when creating a new process, Xinu allocates a stack for the process. Stacks are allocated from the highest address of free space. Second, whenever a process requests heap storage, Xinu allocates the necessary amount of space from the low end of free space. Figure 9.3 shows an example of the memory layout when three processes are executing and heap storage has been allocated.


**Figure 9.3** Illustration of memory after three processes have been created.

†Because physical memory is replicated in the E2100L address space, probing memory is difficult.

## 9.7 Design Of The Low-Level Memory Manager

A set of functions and data structures are used to manage free memory. The low-level memory manager provides four functions:

- *getstk* — allocate stack space when a process is created
- *freestk* — release a stack when a process terminates
- *getmem* — allocate heap storage on demand
- *freemem* — release heap storage as requested

Our design treats free space as a single, exhaustable resource — the low-level memory manager allocates space provided a request can be satisfied. Furthermore, the low-level memory manager does not partition the free space into memory available for process stacks and memory available for heap variables. Requests of one type can take the remaining free space, and leave none for the other type. Of course, such an allocation only works if processes cooperate. Otherwise, a given process can consume all free memory, leaving no space for other processes. Chapter 10 illustrates an alternative approach by describing a set of high-level memory management functions that prevent exhaustion by partitioning memory among subsystems. The high-level memory manager also demonstrates how processes can block until memory becomes available.

Functions *getstk* and *freestk* are not intended for general use. Instead, when it forms a new process, *create* calls *getstk* to allocate a stack. *Getstk* obtains a block of memory from the highest address of free space, and returns a pointer to the block. *Create* records the size and location of the allocated stack space in the process table entry, and places the stack address in a *CONTEXT* area on top of the stack. Later, when the process becomes current, the context switch accesses the *CONTEXT* area and loads the address of the stack into the stack pointer register. Finally, when the process terminates, *kill* calls function *freestk* to release the process's stack and return the block to the free list.

Functions *getmem* and *freemem* perform analogous functions for heap storage. Unlike the stack allocation functions, *getmem* and *freemem* allocate blocks from the lowest address of the free space.

## 9.8 Allocation Strategy And Memory Persistence

Because only *create* and *kill* allocate and free process stacks, the system can guarantee that the stack space allocated to a process will be released when the process exits. However, the system does not record the set of blocks that a process allocates from the heap by calling *getmem*. Therefore, the system does not automatically release heap storage. As a consequence, the burden of returning heap space is left to the programmer:

*Heap space persists independent of the process that allocates the space. Before it exits, a process must explicitly release storage that it has allocated from the heap, or the space will remain allocated.*

Of course, returning allocated heap space does not guarantee that the heap will never be exhausted. On one hand, demand can exceed the available space. On the other hand, the free space can become *fragmented* into small, discontiguous pieces that are each too small to satisfy a request. The next chapter continues the discussion of allocation policies, and shows one approach to avoiding fragmentation of the free space.

## 9.9 Keeping Track Of Free Memory

A memory manager must keep information about all free memory blocks. To do so, the memory manager forms a list, where each item on the list specifies a memory address at which a block starts and a length. Initially, the list contains only one item that corresponds to the block of memory between the end of the program and the highest point in memory. When a process requests a block of memory, the memory manager searches the list, finds a free area, allocates the requested size block, and updates the list to show that more of the free memory has been allocated. Similarly, whenever a process releases a previously allocated block of memory, the memory manager adds the block to the list. Figure 9.4 illustrates an example set of four free memory blocks.

| Block | Address | Length |
|---|---|---|
| 1 | 0x84F800 | 4096 |
| 2 | 0x850F70 | 8192 |
| 3 | 0x8A03F0 | 8192 |
| 4 | 0x8C01D0 | 4096 |

**Figure 9.4** A conceptual list of free memory blocks.

A memory manager must examine each transaction carefully to avoid generating an arbitrarily long list of blocks. When a block is released, the memory manager scans the list to see if the released block is adjacent to the end of one of the existing free blocks. If so, the size of the existing block can be increased without adding a new entry to the list. Similarly, if the new block is adjacent to the beginning of an existing block, the entry can be updated. Finally, if a released block exactly fills the gap between two free blocks on the list, the memory manager will combine the two existing entries on the list into one giant block that covers all the memory from the two on the list plus the released block. We use the term *coalesce* to describe combining entries. The point is: if a memory manager is built correctly, once all allocated blocks have been released, the

free list will be back to the initial state where there is one entry that corresponds to all free memory between the end of the program and the highest memory address.

## 9.10 Implementation Of Low-Level Memory Management

Where should the list of free memory blocks be stored? Our example implementation follows a standard approach by using free memory itself to store the list. After all, a free memory block is not being used, so the contents are no longer needed. Thus, the free memory blocks can be chained together to form a linked list by placing a pointer in each block to the next block.

In the code, global variable *memlist* contains a pointer to the first free block. The key to understanding the implementation lies in the invariant maintained at all times:

> *All free blocks of memory are kept on a linked list; blocks on the free list are ordered by increasing address.*

The conceptual list in Figure 9.4 shows two fields associated with each entry: the address and a size. In our linked list implementation, each node in the list points to (i.e., gives the address) of the next node. However, we must also store the size of each block. Therefore, each block of free memory contains two items: a pointer to the next free block of memory and an integer that gives the size of the current block. Figure 9.5 illustrates the concept.

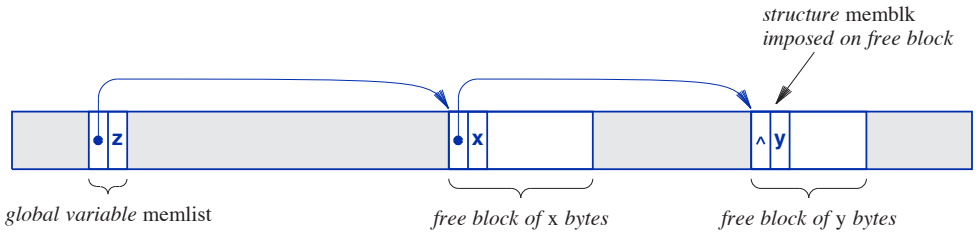



**Figure 9.5** Illustration of a free memory list that contains two memory blocks.

Structure *memblk*, defined in file *memory.h*, gives the shape of a structure that can be imposed on each free node. In struct *memblk*, field *mnext* points to the next block on the list or contains the value *NULL* to indicate that a block is the final block on the list. Field *mlength* specifies the length of the current block in bytes, including the header. Note that the length is declared to be an unsigned long, which accommodates any size block up to the entire physical address space.

Variable *memlist*, which constitutes the head of the list, is defined as consisting of a *memblk* structure. Thus, the head of the list has exactly the same form that we impose on other nodes in the list. However, the *mlength* field (used to store the size of a block) is not meaningful in variable *memlist* because the size of *memlist* is *sizeof(struct memblk)*. Consequently, we can use the length field for another purpose. Xinu uses the field to store the total size of free memory (i.e., a sum of the length field in each block). Having a count of the free memory can help when debugging or to assess whether the system is approaching the maximum possible size.

Note that each block on the free list must hold a complete *memblk* structure (i.e., eight bytes). The design choice has a consequence: a memory manager cannot store a free block of less then eight bytes. How can we guarantee that no process attempts to free a smaller amount of memory? We can tell programmers that they must free exactly the same amount they request, and allow the memory management routines to insure all requests are at least eight bytes. But another problem can arise if the memory manager extracts a piece from a free block: subtraction can leave a remainder of less than eight bytes. To solve the problem, our memory manager rounds all requests to a multiple of *memblk* structures. Two inline functions, *roundmb* and *truncmb*, perform the task. Function *roundmb* rounds requests to multiples of eight bytes, and *truncmb* is used to truncate a memory size to a multiple of eight bytes. Truncation is only used once: the initial size of free space must be truncated rather than rounded. The key point is:

> *Rounding all requests to multiples of the* memblk *structure insures that each request satisfies the constraint and guarantees that no free block will ever be too small to link into the free list.*

File *memory.h* contains declarations related to memory management, including definitions of the two inline functions, *roundmb* and *truncmb*. To make the implementation efficient, the code for the two functions uses constants and Boolean operations rather than the *sizeof* function and division. Using Boolean operations is only possible because the size of a memory block is a power of two.

```
/* memory.h - roundmb, truncmb, freestk */

#define PAGE_SIZE       4096
#define MAXADDR         0x02000000      /* 160NL has 32MB RAM           */

/*------------------------------------------------------------------------
 * roundmb, truncmb - round or truncate address to memory block size
 *------------------------------------------------------------------------
 */
#define roundmb(x)      (char *)( (7 + (uint32)(x)) & (~7) )
#define truncmb(x)      (char *)( ((uint32)(x)) & (~7) )

/*------------------------------------------------------------------------
 *  freestk  --  free stack memory allocated by getstk
 *------------------------------------------------------------------------
 */
#define freestk(p,len)  freemem((char *)((uint32)(p))           \
                                - ((uint32)roundmb(len))        \
                                + (uint32)sizeof(uint32)),      \
                                (uint32)roundmb(len) )

struct  memblk  {                       /* see roundmb & truncmb        */
        struct  memblk  *mnext;         /* ptr to next free memory blk  */
        uint32  mlength;                /* size of blk (includes memblk)*/
        };
extern  struct  memblk  memlist;        /* head of free memory list     */
extern  void    *maxheap;               /* max free memory address      */
extern  void    *minheap;               /* address beyond loaded memory */

/* added by linker */

extern  int     end;                    /* end of program               */
extern  int     edata;                  /* end of data segment          */
extern  int     etext;                  /* end of text segment          */
```

## 9.11 Allocating Heap Storage

Function *getmem* allocates heap storage by finding a free block that is sufficient for the request. Our implementation uses a *first-fit* allocation strategy by allocating the first block on the free list that satisfies a request. *Getmem* subtracts the requested memory from the free block and adjusts the free list accordingly. File *getmem.c* contains the code.

```
/* getmem.c - getmem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getmem  -  Allocate heap storage, returning lowest word address
 *------------------------------------------------------------------------
 */
char    *getmem(
          uint32        nbytes          /* size of memory requested     */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  memblk  *prev, *curr, *leftover;

        mask = disable();
        if (nbytes == 0) {
                restore(mask);
                return (char *)SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* use memblk multiples */

        prev = &memlist;
        curr = memlist.mnext;
        while (curr != NULL) {                  /* search free list     */

                if (curr->mlength == nbytes) {  /* block is exact match */
                        prev->mnext = curr->mnext;
                        memlist.mlength -= nbytes;
                        restore(mask);
                        return (char *)(curr);

                } else if (curr->mlength > nbytes) { /* split big block */
                        leftover = (struct memblk *)((uint32) curr +
                                        nbytes);
                        prev->mnext = leftover;
                        leftover->mnext = curr->mnext;
                        leftover->mlength = curr->mlength - nbytes;
                        memlist.mlength -= nbytes;
                        restore(mask);
                        return (char *)(curr);
                } else {                        /* move to next block   */
                        prev = curr;
                        curr = curr->mnext;
```

```
                }
        }
        restore(mask);
        return (char *)SYSERR;
}
```

After verifying that its argument is valid and the free list is not empty, *getmem* uses *roundmb* to round the memory request to a multiple of *memblk* bytes, and then searches the free list to find the first block of memory large enough to satisfy the request. Because the free list is singly linked, *getmem* uses two pointers, *prev* and *curr*, to walk the list. *Getmem* maintains the following invariant during the search: when *curr* points to a free block, *prev* points to its predecessor on the list (possibly the head of the list, *memlist*). As it runs through the list, the code must insure the invariant remains intact. Consequently, when a free block is discovered that is large enough to satisfy the request, *prev* will point to the predecessor.

At each step, *getmem* compares the size of the current block to *nbytes*, the size of the request. There are three cases. If the size of the current block is less than size requested, *getmem* moves to the next block on the list and continues the search. If the size of the current block exactly matches the size of the request, *getmem* merely removes the block from the free list (by making the *mnext* field in the predecessor block point to the successor block), and returns a pointer to the current block. If the size of the current block is greater than the size requested, *getmem* partitions the current block into two pieces: one of size *nbytes* that will be returned to the caller, and a remaining piece that will be left on the free list. To perform the division, *getmem* computes the address of the remaining piece, and places the address in variable *leftover*. Computing such an address is conceptually simple: the leftover piece lies *nbytes* beyond the beginning of the block. However, adding *nbytes* to pointer *curr* does not produce the desired result because C performs pointer arithmetic. To force C to use integer arithmetic instead of pointer arithmetic, *curr* is changed to an unsigned integer with a cast (i.e., *(uint32)curr*) before adding *nbytes*. Once a sum has been computed, the result is changed back into a pointer to a memory block using another cast. After *leftover* has been computed, the *mnext* field of the *prev* block is updated, and the *mnext* and *mlength* fields in the *leftover* block are assigned.

The code relies on a fundamental mathematical relationship: subtracting two multiples of *K* will produce a multiple of *K*. In the example, *K* is the size of a *memblk* structure. Thus, if the system begins by using *roundmb* to round the size of free memory, and always uses *roundmb* to round requests, each free block and each leftover piece will be large enough to hold a *memblk* structure.

## 9.12 Allocating Stack Storage

Function *getstk* allocates a block of memory for a process stack. We will see that *getstk* is invoked whenever a process is created. The code appears in file *getstk.c*.

Because the free list is kept in order by memory address and stack space is allocated from the highest available block, *getstk* must search the entire list of free blocks. During the search, *getstk* records the address of any block that satisfies the request, which means that after the search completes, the last recorded address points to the free block with the highest address that satisfies the request.† As with *getmem*, *getstk* maintains the invariant that during the search, variables *next* and *prev* point to a free block of memory and the predecessor of the free block, respectively. Whenever a block is found that has sufficient size to satisfy the request, *getstk* sets variable *fits* to the address of the block and sets variable *fitsprev* to the address of predecessor. Thus, when the search completes, *fits* points to the usable free block with the highest memory address (or will remain equal to *NULL* if no block satisfies the request).

Once the search completes and a block has been found, two cases arise, analogous to the cases in *getmem*. If the size of the block on the free list is exactly the size requested, *getstk* unlinks the block from the free list and returns the address of the block to its caller. Otherwise, *getstk* partitions the block into two pieces, allocating a piece of size *nbytes*, and leaves the remainder on the free list. Because *getstk* returns the piece from the highest part of the selected block, the computation differs slightly from that in *getmem*.

†The strategy of allocating the block with the highest address that satisfies a request is known as the *last-fit* strategy.

```
/* getstk.c - getstk */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getstk  -  Allocate stack memory, returning highest word address
 *------------------------------------------------------------------------
 */
char    *getstk(
          uint32        nbytes          /* size of memory requested     */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  memblk  *prev, *curr;   /* walk through memory list     */
        struct  memblk  *fits, *fitsprev; /* record block that fits     */

        mask = disable();
        if (nbytes == 0) {
                restore(mask);
                return (char *)SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* use mblock multiples */

        prev = &memlist;
        curr = memlist.mnext;
        fits = NULL;

        while (curr != NULL) {                  /* scan entire list     */
                if (curr->mlength >= nbytes) {  /* record block address */
                        fits = curr;            /*   when request fits  */
                        fitsprev = prev;
                }
                prev = curr;
                curr = curr->mnext;
        }

        if (fits == NULL) {                     /* no block was found   */
                restore(mask);
                return (char *)SYSERR;
        }
        if (nbytes == fits->mlength) {          /* block is exact match */
                fitsprev->mnext = fits->mnext;
        } else {                                /* remove top section   */
                fits->mlength -= nbytes;
```

```
                fits = (struct memblk *)((uint32)fits + fits->mlength);
        }
        memlist.mlength -= nbytes;
        restore(mask);
        return (char *)((uint32) fits + nbytes - sizeof(uint32));
}
```

## 9.13 Releasing Heap And Stack Storage

Once it finishes using a block of heap storage, a process calls function *freemem* to return the block to the free list, making the memory eligible for subsequent allocation. Because blocks on the free list are kept in order by address, *freemem* uses the block's address to find the correct location on the list. In addition, *freemem* handles the task of *coalescing* the block with adjacent free blocks. There are three cases: the new block of memory can be adjacent to the previous block, adjacent to the succeeding block, or adjacent to both. When any of the three cases occurs, *freemem* combines the new block with adjacent block(s) to form one large block on the free list. Coalescing helps avoid memory fragmentation.

The code for *freemem* can be found in file *freemem.c*. As in *getmem*, two pointers, *prev* and *next*, run through the list of free blocks. *Freemem* searches the list until the address of the block to be returned lies between *prev* and *next*. Once the correct position has been found, the code performs coalescing.

Coalescing is handled in three steps. The code first checks for coalescing with the previous block. That is, *freemem* adds the length of the previous block to the block's address to compute the address one beyond the previous block. *Freemem* compares the result, which is found in variable *top*, to the address of the block being inserted. If the address of the inserted block equals *top*, *freemem* increases the size of the previous block to include the new block. Otherwise, freemem inserts the new block in the list. Of course, if the previous pointer points to the head of the *memlist*, no coalescing can be performed.

Once it has handled coalescing with the previous block, *freemem* checks for coalescing with the next block. Once again, *freemem* computes the address that lies one beyond the current block, and tests whether the address is equal to the address of the next block. If so, the current block is adjacent to the next block, so *freemem* increases the size of the current block to include the next block, and unlinks the next block from the list.

The important point is that *freemem* handles all three special cases:

> *When adding a block to the free list, the memory manager must check to see whether the new block is adjacent to the previous block, adjacent to the next block, or adjacent to both.*

```
/* freemem.c - freemem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  freemem  -  Free a memory block, returning the block to the free list
 *------------------------------------------------------------------------
 */
syscall freemem(
          char          *blkaddr,       /* pointer to memory block      */
          uint32        nbytes          /* size of block in bytes       */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  memblk  *next, *prev, *block;
        uint32  top;

        mask = disable();
        if ((nbytes == 0) || ((uint32) blkaddr < (uint32) minheap)
                          || ((uint32) blkaddr > (uint32) maxheap)) {
                restore(mask);
                return SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* use memblk multiples */
        block = (struct memblk *)blkaddr;

        prev = &memlist;                        /* walk along free list */
        next = memlist.mnext;
        while ((next != NULL) && (next < block)) {
                prev = next;
                next = next->mnext;
        }

        if (prev == &memlist) {         /* compute top of previous block*/
                top = (uint32) NULL;
        } else {
                top = (uint32) prev + prev->mlength;
        }

        /* Insure new block does not overlap previous or next blocks    */

        if (((prev != &memlist) && (uint32) block < top)
            || ((next != NULL)  && (uint32) block+nbytes>(uint32)next)) {
                restore(mask);
```

```
                return SYSERR;
        }

        memlist.mlength += nbytes;

        /* Either coalesce with previous block or add to free list */

        if (top == (uint32) block) {    /* coalesce with previous block */
                prev->mlength += nbytes;
                block = prev;
        } else {                        /* link into list as new node   */
                block->mnext = next;
                block->mlength = nbytes;
                prev->mnext = block;
        }

        /* Coalesce with next block if adjacent */

        if (((uint32) block + block->mlength) == (uint32) next) {
                block->mlength += next->mlength;
                block->mnext = next->mnext;
        }
        restore(mask);
        return OK;
}
```

Because free memory is treated as a single resource that can be used for stacks or heap storage, releasing stack memory follows the same algorithm as releasing heap storage. The only difference between heap and stack allocations arises because *getmem* returns the lowest address of an allocated block and *getstk* returns the highest address. In the current implementation, *freestk* is an inline function that invokes *freemem*. Before calling *freemem*, *freestk* must convert its argument from the highest address in a block to the lowest. The code is found in memory.h.† Although the current implementation uses a single underlying list, keeping *freestk* separate from *freemem* maintains a conceptual distinction, and makes it easier to modify the implementation later. The point is:

> *Although the current implementation uses the same underlying function to release heap and stack storage, having separate system calls for* freestk *and* freemem *maintains the conceptual distinction and makes the system easier to change later.*

†File memory.h can be found on page 148.

## 9.14 Perspective

Although the mechanisms are relatively straightforward, the design of a memory management subsystem reveals one of the most surprisingly subtle problems in operating systems. The problem arises from a fundamental conflict. On one hand, an operating system is designed to run without stopping. Therefore, operating system facilities must be resource-preserving: when a process finishes with a resource, the system must recapture the resource and make it available to other processes. On the other hand, any memory management mechanism that allows processes to allocate and free blocks of arbitrary size is not resource-preserving because memory can become fragmented, with free memory divided into small, discontiguous blocks. Thus, a designer must be aware that the choice represents a tradeoff. Allowing arbitrary-size allocations makes the system easier to use, but also introduces a potential for problems.

## 9.15 Summary

Large operating systems offer complex memory management schemes that allow the demand for memory to exceed the physical memory size. Such systems store information on secondary storage, and move it to main memory when a reference occurs. Sophisticated memory management systems support multiple virtual address spaces that allow each application to address memory starting at location zero; each virtual address space maps to a different region of physical memory, meaning that the system protects a given application from other applications. We use the term *heavyweight process* to refer to an application that runs in a separate address space; a lightweight process abstraction permits one or more processes to run in each virtual space. Paging is the most widely used technology used to provide virtual address spaces and multiplex them onto physical memory.

Small embedded systems usually keep all code and data in physical memory. An image contains three segments: a text segment consisting of compiled code, a data segment that contains initialized data values, and a bss segment that contains uninitialized variables. When a system starts, physical memory not allocated to the three segments is considered free, and a low-level memory manager allocates the free memory on demand.

The low-level memory manager in Xinu maintains a linked list of free memory blocks; both stack and heap storage is allocated from the list as needed. Heap storage is allocated by finding the first free memory block that satisfies the request (i.e., the free block with the lowest address). A request for stack storage is satisfied from the highest free memory block that satisfies the request. Because the list of free memory blocks is singly linked in order by address, allocating stack space requires searching the entire free list.

The low-level memory manager treats free space as an exhaustable resource with no partition between stack and heap storage. Because the memory manager does not contain mechanisms to prevent a process from allocating all free memory, a program-

mer must plan carefully to avoid starvation. A higher-level memory manager discussed in the next chapter illustrates how memory can be partitioned into regions and how a set of processes can block waiting for memory to become available.

## EXERCISES

**9.1** An early version of the low-level memory manager had no provision for returning blocks of memory to the free list. Speculate about embedded systems: are *freemem* and *freestk* necessary? Why or why not?

**9.2** Replace the low-level memory management functions with a set of functions that allocate heap and stack memory permanently (i.e., without providing a mechanism to return storage to a free list). How do the sizes of the new allocation routines compare to the sizes of *getstk* and *getmem*?

**9.3** Does the approach of allocating stack and heap storage from opposite ends of free space help minimize fragmentation? To find out, consider a series of requests that intermix allocation and freeing of stack storage of 1000 bytes and heap storage of 500 bytes. Compare the approach described in the chapter to an approach that allocates stack and heap requests from the same end of free space (i.e., an approach in which all allocation uses *getmem*). Find a sequence of requests that result in fragmentation if stack and heap requests are not allocated from separate ends.

**9.4** Can the memory management system described here allocate arbitrarily small amounts of memory? Why or why not?

**9.5** Many embedded systems go through a prototype stage, in which the system is built on a general platform, and a final stage, in which minimal hardware is designed for the system. In terms of memory management, one question concerns the size of the stack needed by each process. Modify the code to allow the system to measure the maximum stack space used by a process and report the maximum stack size when the process exits.

# Chapter Contents

# 10

# High-Level Memory Management and Virtual Memory

*Yea, from the table of my memory I'll wipe away all fond trivial records.*

— William Shakespeare

## 10.1 Introduction

Previous chapters consider abstractions that an operating system uses to manage computation and I/O. The previous chapter describes a low-level memory management facility that treats memory as an exhaustable resource. The chapter discusses address spaces, program segments, and functions that manage a global free list. Although they are necessary, low-level memory management facilities are not sufficient for all needs.

This chapter completes the discussion of memory management by introducing high-level facilities. The chapter explains the motivation for partitioning memory resources into independent subsets. It presents a high-level memory management mechanism that allows memory to be divided into independent buffer pools, and explains how allocation and use of the memory in a given pool does not affect the use of memory in other pools. The chapter also describes virtual memory, and explains how virtual memory hardware operates.

## 10.2 Partitioned Space Allocation

Functions *getmem* and *freemem* that were described in the previous chapter constitute a basic memory manager. The design places no limit on the amount of memory that a given process can allocate, nor do the functions attempt to divide free space "fairly." Instead, the allocation functions merely honor requests on a first-come-first-served basis until no free memory remains. Once free memory has been exhausted, the functions reject further requests without blocking processes or waiting for memory to be released. Although it is relatively efficient, a global allocation strategy that forces all processes to contend for the same memory can lead to deprivation, a situation in which one or more processes cannot obtain memory because all memory has been consumed. As a consequence, global memory allocation schemes do not work well for all parts of the operating system.

To understand why a system cannot rely on global allocation, consider software for network communication. Packets arrive at random. Because a network application takes time to process a given packet, additional packets may arrive while one is being handled. If each incoming packet is placed in a memory buffer, exhaustive allocation can lead to disaster. Incoming messages pile up waiting to be processed, and each takes memory. In the worst case, all the available space will be allocated to packet buffers, and none will be available for other operating system functions. In particular, if disk I/O uses memory, all disk I/O may stop until memory becomes available. If the application processing network packets attempts to write to a file, *deadlock* can occur: the process blocks waiting for a disk buffer, but all memory is used for network buffers and no network buffer can be released until disk I/O completes.

To prevent deadlocks, higher-level memory management must be designed to partition free memory into independent subsets, and insure that allocation and deallocation of a given subset remains independent from the allocation and deallocation of other subsets. By limiting the amount of memory that can be used for a particular function, the system can guarantee that excessive requests will not lead to global deprivation. Furthermore, the system can assume that memory allocated for a particular function will always be returned, so it can arrange to suspend processes until their memory request can be satisfied, eliminating the overhead introduced by busy waiting. Partitioning cannot guarantee that no deadlocks will occur, but it does limit unintentional deadlocks that arise when one subsystem takes memory needed by another subsystem.

## 10.3 Buffer Pools

The mechanism we have chosen to handle partitioned memory is known as a *buffer pool* manager. Memory is divided into a set of buffer pools. Each buffer pool contains a fixed number of memory blocks, and all blocks in a given pool are the same size. The term *buffer* was chosen to reflect the intended use in I/O routines and communication software (e.g., disk buffers or buffers for network packets).

The memory space for a particular set of buffers is allocated when the pool is created; once a pool has been allocated, there is no way to increase the number of buffers in the pool or to change the buffer size.

Each buffer pool is identified by an integer, known as a *pool identifier* or *buffer pool ID*. Like other IDs in Xinu, a buffer pool ID is used as an index into the buffer pool table, *buftab*. Once a pool has been created, a process uses the pool ID whenever it requests a buffer from a pool or releases a previously allocated buffer back to a pool. Requests to allocate or release a buffer from a pool do not need to specify the length of a buffer because the size of buffers is fixed when the pool is created.

The buffer pool mechanism differs from the low-level memory manager in another way: the mechanism is *synchronous*. That is, a process that requests a buffer will be blocked until the request can be satisfied. As in many of the previous examples, the implementation uses semaphores to control access to a buffer pool. Each buffer pool has a semaphore, and the code that allocates a buffer calls *wait* on a pool's semaphore. The call returns immediately if buffers remain in the pool, but blocks the caller if no buffers remain. Eventually, when another process returns a buffer to a pool, the semaphore is signaled, which allows a waiting process to obtain the buffer and resume execution.

The data structure used to hold information about buffer pools consists of a single table. Each entry in the table holds a buffer size, a semaphore ID, and a pointer to a linked list of buffers for the pool. Pertinent declarations can be found in file *bufpool.h*:

```
/* bufpool.h */

#ifndef NBPOOLS
#define NBPOOLS 20              /* Maximum number of buffer pools       */
#endif

#ifndef BP_MAXB
#define BP_MAXB 8192            /* Maximum buffer size in bytes         */
#endif

#define BP_MINB 8               /* Minimum buffer size in bytes         */
#ifndef BP_MAXN
#define BP_MAXN 2048            /* Maximum number of buffers in a pool  */
#endif

struct  bpentry {               /* Description of a single buffer pool  */
        struct  bpentry *bpnext;/* pointer to next free buffer          */
        sid32   bpsem;          /* semaphore that counts buffers        */
                                /*    currently available in the pool   */
        uint32  bpsize;         /* size of buffers in this pool         */
        };

extern  struct  bpentry buftab[];/* Buffer pool table                   */
extern  bpid32  nbpools;        /* current number of allocated pools    */
```

Structure *bpentry* defines the contents of an entry in the buffer pool table, *buftab*. The buffers for a given pool are linked into a list, with field *bpnext* pointing to the first buffer on the list. Semaphore *bpsem* controls allocation from the pool, and integer *bpsize* gives the length of buffers in the pool.

## 10.4 Allocating A Buffer

Three functions provide an interface to buffer pools. A process calls function *mkpool* to create a buffer pool and obtain an ID. Once a pool as been created, a process can call function *getbuf* to obtain a buffer, and function *freebuf* to release a buffer back to the pool.

*Getbuf* works as expected, waiting on the semaphore until a buffer is available, and then unlinking the first buffer from the list. The code is found in file *getbuf.c*:

```
/* getbuf.c - getbuf */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getbuf  --  get a buffer from a preestablished buffer pool
 *------------------------------------------------------------------------
 */
char    *getbuf(
          bpid32        poolid          /* index of pool in buftab       */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  bpentry *bpptr;         /* pointer to entry in buftab   */
        struct  bpentry *bufptr;        /* pointer to a buffer          */

        mask = disable();

        /* Check arguments */

        if ( (poolid < 0  ||  poolid >= nbpools) ) {
                restore(mask);
                return (char *)SYSERR;

        }
        bpptr = &buftab[poolid];

        /* Wait for pool to have > 0 buffers and allocate a buffer */

        wait(bpptr->bpsem);
        bufptr = bpptr->bpnext;

        /* Unlink buffer from pool */

        bpptr->bpnext = bufptr->bpnext;

        /* Record pool ID in first four bytes of buffer and skip */

        *(bpid32 *)bufptr = poolid;
        bufptr = (struct bpentry *)(sizeof(bpid32) + (char *)bufptr);
        restore(mask);
        return (char *)bufptr;
}
```

Observant readers may have noticed that *getbuf* does not return the address of the buffer to its caller. Instead, *getbuf* stores the pool ID in the first four bytes of the allocated space, and returns the address just beyond the ID. From a caller's point of view, a call to *getbuf* returns the address of a buffer; the caller does not need to worry that earlier bytes hold the pool ID. The system is transparent: when the pool is created, extra space is allocated in each buffer to hold the pool ID. When a buffer is released, *freebuf* uses the hidden pool ID to determine the pool to which a buffer belongs. The technique of using hidden information to identify a buffer pool turns out to be especially useful when buffers are returned by a process other than the one that allocated the buffer.

## 10.5 Returning Buffers To The Buffer Pool

Function *freebuf* returns a buffer to the pool from which it was allocated. The code is found in file *freebuf.c*:

```
/* freebuf.c - freebuf */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  freebuf  --  free a buffer that was allocated from a pool by getbuf
 *------------------------------------------------------------------------
 */
syscall freebuf(
          char          *bufaddr        /* address of buffer to return  */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  bpentry *bpptr;         /* pointer to entry in buftab   */
        bpid32  poolid;                 /* ID of buffer's pool          */

        mask = disable();

        /* Extract pool ID from integer prior to buffer address */

        bufaddr -= sizeof(bpid32);
        poolid = *(bpid32 *)bufaddr;
        if (poolid < 0  ||  poolid >= nbpools) {
                restore(mask);
                return SYSERR;
        }
```

```
        /* Get address of correct pool entry in table */

        bpptr = &buftab[poolid];

        /* Insert buffer into list and signal semaphore */

        ((struct bpentry *)bufaddr)->bpnext = bpptr->bpnext;
        bpptr->bpnext = (struct bpentry *)bufaddr;
        signal(bpptr->bpsem);
        restore(mask);
        return OK;
}
```

Recall that when it allocates a buffer, *getbuf* stores the pool ID in the four bytes that precede the buffer address. *Freebuf* moves back four bytes from the beginning of the buffer, and extracts the pool ID. After verifying that the pool ID is valid, *getbuf* uses the ID to locate the entry in the table of buffer pools. It then links the buffer back into the linked list of buffers, and signals the pool semaphore *bpsem*, allowing other processes to use the buffer.

## 10.6 Creating A Buffer Pool

Function *mkbufpool* creates a new buffer pool and returns its ID. *Mkbufpool* takes two arguments: the size of buffers in the pool and the number of buffers.

```
/* mkbufpool.c - mkbufpool */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  mkbufpool  --  allocate memory for a buffer pool and link the buffers
 *------------------------------------------------------------------------
 */
bpid32  mkbufpool(
          int32         bufsiz,         /* size of a buffer in the pool */
          int32         numbufs         /* number of buffers in the pool*/
        )
{
        intmask mask;                   /* saved interrupt mask         */
        bpid32  poolid;                 /* ID of pool that is created   */
        struct  bpentry *bpptr;         /* pointer to entry in buftab   */
        char    *buf;                   /* pointer to memory for buffer */

        mask = disable();
        if (bufsiz<BP_MINB || bufsiz>BP_MAXB
            || numbufs<1 || numbufs>BP_MAXN
            || nbpools >= NBPOOLS) {
                restore(mask);
                return (bpid32)SYSERR;
        }
        /* Round request to a multiple of 4 bytes */

        bufsiz = ( (bufsiz + 3) & (~3) );

        buf = (char *)getmem( numbufs * (bufsiz+sizeof(bpid32)) );
        if ((int32)buf == SYSERR) {
                restore(mask);
                return (bpid32)SYSERR;
        }
        poolid = nbpools++;
        bpptr = &buftab[poolid];
        bpptr->bpnext = (struct bpentry *)buf;
        bpptr->bpsize = bufsiz;
        if ( (bpptr->bpsem = semcreate(numbufs)) == SYSERR) {
                nbpools--;
                restore(mask);
                return (bpid32)SYSERR;
        }
        bufsiz+=sizeof(bpid32);
        for (numbufs-- ; numbufs>0 ; numbufs-- ) {
```

```
                bpptr = (struct bpentry *)buf;
                buf += bufsiz;
                bpptr->bpnext = (struct bpentry *)buf;
        }
        bpptr = (struct bpentry *)buf;
        bpptr->bpnext = (struct bpentry *)NULL;
        restore(mask);
        return poolid;
}
```

*Mkbufpool* begins by checking its arguments. If the buffer size is out of range, the requested number of buffers is negative, or the buffer pool table is full, *mkbufpool* reports an error. *Mkbufpool* computes the size of memory required to hold the buffers, and calls *getmem* to allocate the needed memory. If the memory allocation succeeds, *mkbufpool* allocates an entry in the buffer pool table, and fills in entries. It creates a semaphore, saves the buffer size, and stores the address of the allocated memory in the linked list pointer, *bpnext*.

Once the table entry has been initialized, *mkbufpool* iterates through the allocated memory, dividing the block into a set of buffers. It links each buffer onto the free list. Note that when *mkbufpool* creates the free list, each block of memory contains the size of the buffer the user requested plus the size of a buffer pool ID (four bytes). Thus, after the buffer pool ID is stored in the block, sufficient bytes remain for the buffer that the user requested. After the free list has been formed, *mkbufpool* returns the pool ID to its caller.

## 10.7 Initializing The Buffer Pool Table

Function *bufinit* initializes the buffer pool table. The code, found in file *bufinit.c*, is trivial:

```
/* bufinit.c - bufinit */

#include <xinu.h>

struct  bpentry buftab[NBPOOLS];                /* buffer pool table            */
bpid32  nbpools;

/*------------------------------------------------------------------------
 *  bufinit  --  initialize the buffer pool data structure
 *------------------------------------------------------------------------
 */
status  bufinit(void)
{
        nbpools = 0;
        return OK;
}
```

All that *bufinit* needs to do is set the global count of allocated buffer pools. In the example code, buffer pools can be allocated dynamically. However, once it has been allocated, a pool cannot be deallocated. An exercise suggests extending the mechanism to permit dynamic deallocation.

## 10.8 Virtual Memory And Memory Multiplexing

Most large computer systems virtualize memory and present an application with an idealized view. Each application appears to have a large memory that can exceed the size of physical memory. The operating system multiplexes physical memory among all processes that need to use it, moving all or part of the application into physical memory as needed. That is, code and data for processes are kept on secondary storage (i.e., disk), and moved into main memory temporarily when the process is executing. Although few embedded systems rely on virtual memory, many processors include virtual memory hardware.

The chief design question in virtual memory management systems concerns the form of multiplexing. Several possibilities have been used:

- Swapping
- Segmentation
- Paging

*Swapping* refers to an approach that moves all code and data associated with a particular computation into main memory when the scheduler makes the computation current. Swapping works best for a long-running computation, such as a word proces-

sor that runs while a human types a document — the computation is moved into main memory, and remains resident for a long period.

*Segmentation* refers to an approach that moves pieces of the code and data associated with a computation into main memory as needed. One can imagine, for example, placing each function and the associated variables in a separate segment. When a function is called, the operating system loads the segment containing the function into main memory. Seldom-used functions (e.g., a function that displays an error message) remain on secondary storage. In theory, segmentation uses less memory than swapping because segmentation allows pieces of a computation to be loaded into memory as needed. Although the approach has intuitive appeal, few operating systems use dynamic segmentation.

*Paging* refers to an approach that divides each program into small, fixed-size pieces called *pages*. The system keeps the most recently referenced pages in main memory, and moves copies of other pages to secondary storage. Pages are fetched on demand — when a running program references memory location $i$, the memory hardware checks to see whether the page containing location $i$ is *resident* (i.e., currently in memory). If the page is not resident, the operating system suspends the process (allowing other processes to execute), and issues a request to the disk to obtain a copy of the needed page. Once the page has been placed in main memory, the operating system makes the process ready. When the process retries the reference to location $i$, the reference succeeds.

## 10.9 Real And Virtual Address Spaces

In many operating systems, the memory manager supplies each computation with an independent *address space*. That is, an application is given a private set of memory locations that are numbered $0$ through $M-1$. The operating system works with the underlying hardware to map each address space to a set of memory locations in memory. As a result, when an application references address zero, the reference is mapped to the memory location that corresponds to zero for the process. When another application references address zero, the reference is mapped to a different location. Thus, although multiple applications can reference address zero, each reference maps to a separate location, and the applications do not interfere with one another. To be precise, we use the term *physical address space* or *real address space* to define the set of addresses that the memory hardware provides, and the term *virtual address space* to describe the set of addresses available to a given computation. A memory manager maps one or more virtual address spaces onto the underlying physical address space. For example, Figure 10.1 illustrates how three virtual address spaces of $K$ locations can be mapped onto an underlying physical address space that contains $3K$ locations.

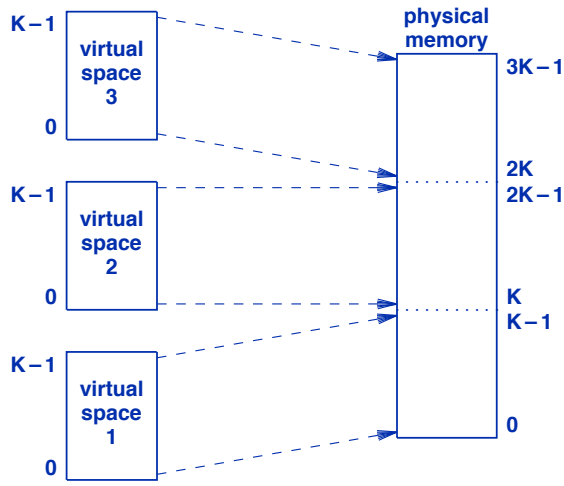



**Figure 10.1** Illustration of three virtual address spaces mapped onto a single underlying physical address space.

From the point of view of a running computation, only addresses in the virtual address space can be referenced. Furthermore, because the system maps each address from a given virtual space to a specific region of memory, the running computation cannot accidentally read or overwrite memory that has been allocated to another computation. As a result, a system that provides each computation with its own virtual address space helps detect programming errors and prevent problems. The point can be summarized:

> *A memory management system that maps each virtual address space into a unique block of memory prevents one computation from reading or writing memory allocated to another computation.*

In Figure 10.1, each virtual address space is smaller than the underlying physical address space. However, most memory management systems permit a virtual address space to be larger than the memory on the machine. For example, a demand paging system only keeps pages that are being referenced in main memory, and leaves copies of other pages on disk.

## 10.10 Hardware For Demand Paging

An operating system that maps between virtual and real addresses cannot operate without hardware support. To understand why, observe that each address, including addresses generated at run-time, must be mapped. Thus, if a program computes a value *C* and then *jumps* to location *C*, the memory system must map *C* to the corresponding real memory address. Only a hardware unit can perform the mapping efficiently.

The hardware needed for demand paging consists of a page table and an address translation unit. A page table resides in kernel memory, and there is one page table for each process. Typically, the hardware contains a register that points to the current page table and a second register that specifies the length; after it has created a page table in memory, the operating system assigns values to the registers and turns on demand paging. Similarly, when a context switch occurs, the operating system changes the page table registers to point to the page table for the new process. Figure 10.2 illustrates the arrangement.

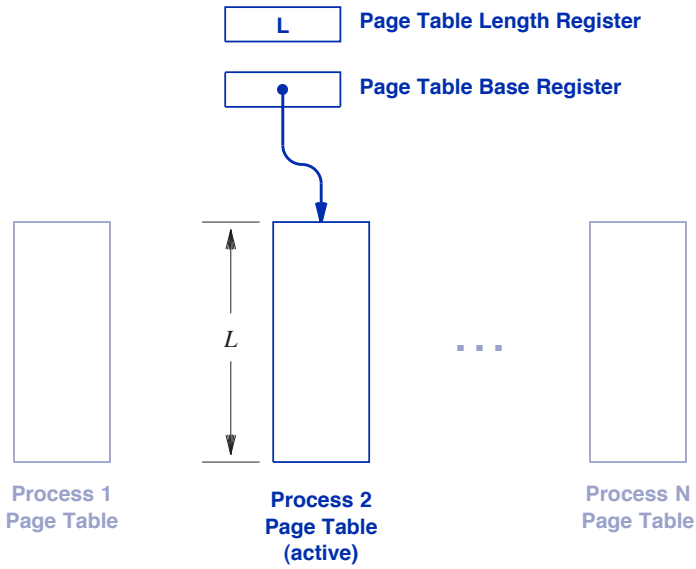



**Figure 10.2** Page tables in memory and hardware registers that specify which page table to use at a given time.

## 10.11 Address Translation With A Page Table

Conceptually, a page table consists of an array of pointers to memory locations. In addition to a pointer, each entry contains a bit that specifies whether the entry is valid (i.e., whether it has been initialized). *Address translation* hardware uses the current

page table to translate a memory address; translation is applied to the instruction addresses as well as addresses used to fetch or store data. Translation consists of array lookup: the hardware treats the high-order bits of an address as a *page number*, uses the page number as an index into the page table, and follows the pointer to the location of the page in memory.

In practice, a page table entry does not contain a complete memory pointer. Instead, pages are restricted to start in memory locations that have zeroes in the low-order bits, and the low-order bits are omitted from each page table entry. For example, suppose a computer has 32-bit addressing and uses 4096-byte pages (i.e., each page contains $2^{12}$ bytes). If memory is divided into a set of 4096-byte *frames*, the starting address of each frame (i.e., the address of the first byte in the frame) will have zeroes in the 12 low-order bits. Therefore, to point to a frame in memory, a page table entry only needs to contain the upper 20 bits.

To translate an address, *A*, the hardware uses the upper bits of *A* as an index into the page table, extracts the address of a frame in memory where the page resides, and then uses the low-order bits of *A* as an offset into the frame. We can imagine that the translation forms a physical memory address as Figure 10.3 illustrates.

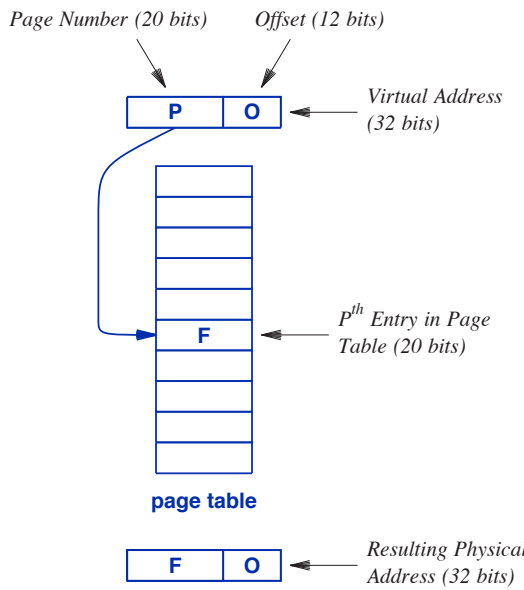



**Figure 10.3** An example of virtual address translation used with paging.

Our description implies that each address translation requires a page table access (i.e., a memory access). However, such overhead would be intolerable. To make translation efficient, a processor employs a special-purpose hardware unit known as a *Translation Look-aside Buffer* (*TLB*). The TLB caches recently accessed page table entries

and can look up the answer much faster than a conventional memory access.† With a TLB, a processor can operate as fast with address translation as it can with address translation disabled.

## 10.12 Metadata In A Page Table Entry

In addition to a frame pointer, each page table entry contains three bits of metadata that the hardware and operating system use. Figure 10.4 lists the bits and their meaning.

| Name | Meaning |
|---|---|
| Use Bit | Set by hardware whenever the page is referenced (fetch or store) |
| Modify Bit | Set by hardware whenever a store operation changes data on the page |
| Presence Bit | Set by OS to indicate whether the page is resident in memory |

**Figure 10.4** The three metabits in each page table entry and their meanings.

## 10.13 Demand Paging And Design Questions

The term *demand paging* refers to a system where an operating system places the pages for all processes on disk, and only reads a page into memory when the page is needed (i.e., on demand). Special processor hardware is needed to support demand paging: if a process attempts to access a page that is not resident in memory, the hardware must suspend execution of the current instruction and notify the operating system by signaling a *page fault* exception. When a page fault occurs, the operating system finds an unused frame in memory, reads the needed page from disk, and then instructs the processor to resume the instruction that caused the fault.

When a computer first starts, memory is relatively empty, which makes finding a free frame easy. Eventually, however, all frames in memory will be filled and the operating system must select one of the filled frames, copy the page back to disk (if the page has been modified), fetch the new page, and change the page tables accordingly. The selection of a page to move back to disk forms a key problem for operating systems designers.

The questions surrounding paging design center on the relationship of pages and processes. When process *X* encounters a page fault, should the operating system move one of process *X*'s pages back to disk or should the system select a page from another process? While a page is being fetched from disk, the operating system can run another

†To achieve high speed, a TLB uses *Content-Addressable Memory* (*CAM*).

process. How can the operating system insure that at least some process has enough pages to run without also generating a page fault?† Should some pages be locked in memory? If so, which ones? How will the page selection policy interact with other policies, such as scheduling? For example, should the operating system guarantee each high-priority process a minimum number of resident pages? If the system allows processes to share memory, what policies apply to the shared pages?

An interesting tradeoff arises in the design of a paging system. Paging overhead and the latency a process experiences can be reduced by giving a process the maximal amount of physical memory when the process runs. However, many processes are I/O-bound, which means that a given process is likely to block waiting for I/O. When one process blocks, overall performance is maximized if another process is ready to execute and the operating system can switch context. That is, CPU utilization and overall throughput of a system can be increased by having many processes ready to execute. So the question arises: should a given process be allowed to use many frames of memory or should memory be divided among processes?

## 10.14 Page Replacement And Global Clock

Various page replacement policies have been proposed and tried:

- Least Recently Used (LRU)
- Least Frequently Used (LFU)
- First In First Out (FIFO)

Interestingly, a provably optimal replacement policy has been discovered. Known as *Belady's optimal page replacement algorithm*, the policy chooses to replace the page that will be referenced farthest in the future. Of course, the method is totally impractical because the operating system cannot know how pages will be used in the future. However, Belady's algorithm allows researchers to assess how well replacement policies perform.

In terms of practical systems, a single algorithm has become the de facto standard for page replacement. Known as *global clock* or *second chance*, the algorithm was devised as part of the MULTICS operating system and has relatively low overhead. The term *global* means that all processes compete with one another (i.e., when process *X* generates a page fault, the operating system can choose a frame from another process, *Y*). The alternative name of the algorithm arises because global clock is said to give used frames a "second chance" before they are reclaimed.

Global clock starts running whenever a page fault occurs. The algorithm maintains a pointer that sweeps through all the frames in memory, stopping when a free frame is found. The next time it runs, the algorithm starts at the frame just beyond where it left off.

To determine whether to select a frame, global clock checks the Use/Modify bits in the page table of the frame. If the Use/Modify bits have value (0,0), global clock chooses the frame. If the bits are (1,0), global clock resets them to (0,0) and by-

†If insufficient frames exist in memory, a paging system can *thrash*, which means the frequency of page faults becomes so high that each the system spends all its time paging and each process spends long periods waiting for pages.

passes the frame. Finally, if the bits are (1,1), global clock changes them to (1,0) and bypasses the frame, keeping a copy of the modified bit to know whether the page has been modified. In the worst case, global clock sweeps through all frames twice before reclaiming one.

In practice, most implementations use a separate process to run the global clock algorithm (which allows the clock to perform disk I/O). Furthermore, global clock does not stop immediately once a frame has been found. Instead, the algorithm continues to run, and collects a small set of candidate pages. Collecting a set allows subsequent page faults to be handled quickly and avoids the overhead associated with running the global clock algorithm frequently (i.e., avoids frequent context switching).

## 10.15 Perspective

Although address space management and virtual memory subsystems comprise many lines of code in an operating system, the most significant intellectual aspects of the problem arise from the choice of allocation policies and the consequent tradeoffs. Allowing each subsystem to allocate arbitrary amounts of memory maximizes flexibility and avoids the problem of a subsystem being deprived when free memory exists. Partitioning memory maximizes protection and avoids the problem of having one subsystem deprive other subsystems. Thus, a tradeoff exists between flexibility and protection.

Despite years of research, no general solution has emerged, the tradeoffs have not been quantified, and no general guidelines exist. Similarly, despite years of research on virtual memory systems, no demand paging algorithms exist that work well for small memories. Fortunately, economics and technology have made many of the problems associated with memory management irrelevant: DRAM chip density increased rapidly, making huge memories inexpensive. As a result, computer vendors avoid memory management altogether by making the memory on each new product so much larger than the memory on the previous product that operating systems never need to invoke demand paging.

## 10.16 Summary

Low-level memory allocation mechanisms treat all of free memory as a single, exhaustible resource. High-level memory management facilities that allow memory to be partitioned into separate regions provide guarantees that prevent one subsystem from using all available memory.

The high-level memory management functions in Xinu use a buffer pool paradigm in which a fixed set of buffers is allocated in each pool. Once a pool has been created, a group of processes can allocate and free buffers dynamically. The buffer pool interface is synchronous: a given process will block until a buffer becomes available.

Large operating systems use virtual memory mechanisms to allocate separate address spaces for application processes. The most popular virtual memory mechanism, paging, divides the address space into fixed-size pages, and loads pages on demand. Hardware is needed to support paging because each memory reference must be mapped from a virtual address to a corresponding physical address.

## EXERCISES

**10.1** Design a new *getmem* that subsumes *getbuf*. Hint: allow the user to suballocate from a previously allocated block of memory.

**10.2** *Mkbufpool* forms a linked list of all buffers in a pool. Explain how to modify the code so it allocates memory but does not link buffers together until a call to *getbuf* requires a new buffer to be allocated.

**10.3** Is *freebuf* more efficient than *freemem*? Justify your answer.

**10.4** Revise the buffer pool allocation mechanism to allow deallocation of buffer pools.

**10.5** The current implementation of buffer pools hides a pool ID in memory just prior to the buffer. Rewrite *freebuf* to eliminate the need for a pool ID. Insure your version of *freebuf* will detect an invalid address (i.e., will not return a buffer to a pool unless the buffer was previously allocated from the pool).

**10.6** Suppose a processor has support for paging. Describe paging hardware that can be used to protect a process's stack from access by other processes, even if demand paging is not implemented (i.e., all pages remain resident and no replacement is performed).

**10.7** Implement the scheme devised in the previous exercise to protect stacks in Xinu.

# NOTES

# Chapter Contents

# 11

# High-Level Message Passing

*I'm always uneasy with messages.*

— Neil Tennant

## 11.1 Introduction

Chapter 8 describes a low-level message passing facility that permits a process to pass a message directly to another process. Although it provides a useful function, the low-level message passing system cannot be used to coordinate multiple receivers, nor can a given process participate in several message exchanges without interference among them.

This chapter completes the discussion of message passing by introducing a high-level facility that provides a synchronous interface for buffered message exchange. The message mechanism allows an arbitrary subset of processes to pass messages without affecting other processes. It introduces the concept of named rendezvous points that exist independent of processes. The implementation uses the buffer pool mechanism from the previous chapter.

## 11.2 Inter-Process Communication Ports

Xinu uses the term *inter-process communication port* to refer to a rendezvous point through which processes can exchange messages. Message passing through ports differs from process-to-process message passing described in Chapter 8 because ports

allow multiple outstanding messages, and processes accessing them are blocked until requests can be satisfied. Each port is configured to hold up to a specified number of messages; each message occupies a 32-bit word. Once it has produced a message, a process can use function *ptsend* to send the message to a port. Messages are deposited in a port in FIFO order. Once a message has been sent, the sending process can continue to execute. At any time, a process can call function *ptrecv* to receive the next message from a port.

Both message sending and receiving are synchronous. As long as space remains in a port, a sender can deposit a message with no delay. Once a port becomes full, however, each sending process is blocked until a message has been removed and space becomes available. Similarly, if a process tries to receive a message from an empty port, the process will be blocked until a message arrives. Requests are also handled in FIFO order. For example, if multiple processes are waiting on an empty port, the process that has been waiting the longest will receive the message. Similarly, if multiple processes are blocked waiting to send, the process that has been waiting the longest is allowed to proceed when a space becomes available.

## 11.3 The Implementation Of Ports

Each port consists of a queue to hold messages and two semaphores. One of the semaphores controls producers, blocking any process that attempts to add messages to a full port. The other semaphore controls consumers, blocking any process that attempts to remove a message from an empty port.

Because ports can be created dynamically, it is impossible to know the total count of items that will be enqueued at all ports at any given time. Although each message is small (one word), the total space required for port queues must be limited to prevent the port functions from using all the free space. To guarantee a limit on the total space used, the port functions allocate a fixed number of nodes to hold messages, and share the set among all ports. Initially, the message nodes are linked into a free list given by variable *ptfree*. Function *ptsend* removes a node from the free list, stores the message in the node, and adds the node to the queue associated with the port to which the message has been sent. Function *ptrecv* extracts the next message from a specified port, returns the node containing the message to the free list, and delivers the message to its caller.

In file *ports.h*, structure *ptnode* defines the contents of a node that contains one message. The two fields in *ptnode* are expected: *ptmsg* holds a 32-bit message, and *ptnext* points to the next message.

Structure *ptentry* defines the contents of an entry in the port table. Fields *ptssem* and *ptrsem* contain the IDs of semaphores that control sending and receiving. Field *ptstate* specifies whether the entry is currently being used, and field *ptmaxcnt* specifies the maximum messages that are allowed in the port at any time. Fields *pthead* and *pttail* point to the first node on the message list and the last, respectively. We will discuss the sequence field, *ptseq*, later.

```
/* ports.h - isbadport */

#define NPORTS          30              /* maximum number of ports      */
#define PT_MSGS         100             /* total messages in system     */
#define PT_FREE         1               /* port is free                 */
#define PT_LIMBO        2               /* port is being deleted/reset  */
#define PT_ALLOC        3               /* port is allocated            */

struct  ptnode  {                       /* node on list of messages     */
        uint32  ptmsg;                  /* a one-word message           */
        struct  ptnode  *ptnext;        /* ptr to next node on list     */
};

struct  ptentry {                       /* entry in the port table      */
        sid32   ptssem;                 /* sender semaphore             */
        sid32   ptrsem;                 /* receiver semaphore           */
        uint16  ptstate;                /* port state (FREE/LIMBO/ALLOC)*/
        uint16  ptmaxcnt;               /* max messages to be queued    */
        int32   ptseq;                  /* sequence changed at creation */
        struct  ptnode  *pthead;        /* list of message pointers     */
        struct  ptnode  *pttail;        /* tail of message list         */
};

extern  struct  ptnode  *ptfree;        /* list of free nodes           */
extern  struct  ptentry porttab[];      /* port table                   */
extern  int32   ptnextid;               /* next port ID to try when     */
                                        /*   looking for a free slot    */

#define isbadport(portid)       ( (portid)<0 || (portid)>=NPORTS )
```

## 11.4 Port Table Initialization

Because initialization code is designed after basic operations have been implemented, we have been discussing initialization functions after other functions. In the case of ports, however, we will discuss initialization first, because doing so will make the remaining functions easier to understand. File *ptinit.c* contains the code to initialize ports along with declaration of the port table. Global variable *ptnextid* gives the index in array *porttab* at which the search will start when a new port is needed. Initialization consists of marking each port free, forming the linked list of free nodes, and initializing *ptnextid*. To create a free list, *ptinit* uses *getmem* to allocate a block of memory, and then moves through the memory, linking individual message nodes together to form a free list.

```
/* ptinit.c - ptinit */

#include <xinu.h>

struct  ptnode  *ptfree;                /* list of free message nodes   */
struct  ptentry porttab[NPORTS];        /* port table                   */
int32   ptnextid;                       /* next table entry to try      */

/*------------------------------------------------------------------------
 *  ptinit  --  initialize all ports
 *------------------------------------------------------------------------
 */
syscall ptinit(
          int32         maxmsgs         /* total messages in all ports  */
        )
{
        int32   i;                      /* runs through port table      */
        struct  ptnode  *next, *prev;   /* used to build free list      */

        ptfree = (struct ptnode *)getmem(maxmsgs*sizeof(struct ptnode));
        if (ptfree == (struct ptnode *)SYSERR) {
                panic("pinit - insufficient memory");
        }

        /* Initialize all port table entries to free */

        for (i=0 ; i<NPORTS ; i++) {
                porttab[i].ptstate = PT_FREE;
                porttab[i].ptseq = 0;
        }
        ptnextid = 0;

        /* Create free list of message pointer nodes */

        for ( prev=next=ptfree ;  --maxmsgs > 0  ; prev=next )
                prev->ptnext = ++next;
        prev->ptnext = NULL;
        return(OK);
}
```

## 11.5 Port Creation

Port creation consists of allocating an entry in the port table from among those that are free. Function *ptcreate* performs the allocation and returns a *port identifier* (*port ID*) to its caller. *Ptcreate* takes an argument that specifies the maximum count of outstanding messages that the port will allow. Thus, when a port is created, the calling process can determine how many messages can be enqueued on the port before any sender blocks.

```
/* ptcreate.c - ptcreate */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptcreate  --  create a port that allows "count" outstanding messages
 *------------------------------------------------------------------------
 */
syscall ptcreate(
          int32         count
        )
{
        intmask mask;                   /* saved interrupt mask         */
        int32   i;                      /* counts all possible ports    */
        int32   ptnum;                  /* candidate port number to try */
        struct  ptentry *ptptr;         /* pointer to port table entry  */

        mask = disable();
        if (count < 0) {
                restore(mask);
                return(SYSERR);
        }

        for (i=0 ; i<NPORTS ; i++) {    /* count all table entries      */
                ptnum = ptnextid;       /* get an entry to check        */
                if (++ptnextid >= NPORTS) {
                        ptnextid = 0;   /* reset for next iteration     */
                }

                /* Check table entry that corresponds to ID ptnum */

                ptptr= &porttab[ptnum];
                if (ptptr->ptstate == PT_FREE) {
                        ptptr->ptstate = PT_ALLOC;
                        ptptr->ptssem = screate(count);
```

```
                        ptptr->ptrsem = screate(0);
                        ptptr->pthead = ptptr->pttail = NULL;
                        ptptr->ptseq++;
                        ptptr->ptmaxcnt = count;
                        restore(mask);
                        return(ptnum);
                }
        }
        restore(mask);
        return(SYSERR);
}
```

## 11.6 Sending A Message To A Port

The basic operations on ports, sending and receiving messages, are handled by functions *ptsend* and *ptrecv*. Each requires the caller to specify the port on which the operation is to be performed by passing a port ID as an argument. Function *ptsend* adds a message to those that are waiting at the port. It waits for space in the port, enqueues the message given by its argument, signals the receiver semaphore to indicate another message is available, and returns. The code is found in file *ptsend.c*:

```
/* ptsend.c - ptsend */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptsend  --  send a message to a port by adding it to the queue
 *------------------------------------------------------------------------
 */
syscall ptsend(
          int32         portid,         /* ID of port to use            */
          umsg32        msg             /* message to send              */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  ptentry *ptptr;         /* pointer to table entry       */
        int32   seq;                    /* local copy of sequence num.  */
        struct  ptnode  *msgnode;       /* allocated message node       */
        struct  ptnode  *tailnode;      /* last node in port or NULL    */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
```

```
                restore(mask);
                return SYSERR;
        }

        /* Wait for space and verify port has not been reset */

        seq = ptptr->ptseq;             /* record orignal sequence      */
        if (wait(ptptr->ptssem) == SYSERR
            || ptptr->ptstate != PT_ALLOC
            || ptptr->ptseq != seq) {
                restore(mask);
                return SYSERR;
        }
        if (ptfree == NULL) {
                panic("Port system ran out of message nodes");
        }

        /* Obtain node from free list by unlinking */

        msgnode = ptfree;               /* point to first free node */
        ptfree  = msgnode->ptnext;      /* unlink from the free list*/
        msgnode->ptnext = NULL;         /* set fields in the node   */
        msgnode->ptmsg  = msg;

        /* Link into queue for the specified port */

        tailnode = ptptr->pttail;
        if (tailnode == NULL) {         /* queue for port was empty */
                ptptr->pttail = ptptr->pthead = msgnode;
        else {                          /* insert new node at tail  */
                tailnode->ptnext = msgnode;
                ptptr->pttail = msgnode;
        }
        signal(ptptr->ptrsem);
        restore(mask);
        return OK;
}
```

The initial code in *ptsend* merely verifies that argument *portid* specifies a valid port ID. What happens next is more interesting. *Ptsend* makes a local copy of the sequence number, *ptseq*, and then processes the request. It waits on the sender semaphore, and then verifies that the port is still allocated and that the sequence number agrees. It may seem odd that *ptsend* verifies the port ID a second time. However, if the port is already full when *ptsend* runs, the calling process can be blocked. Further-

more, while the process is blocked waiting to send, the port may be deleted (and even recreated). To understand how the sequence number helps, recall that *ptcreate* increments the sequence number when a port is created. The idea is to have waiting processes verify that the *wait* did not terminate because the port was deleted. If it did, the port will either remain unused or the sequence number will be incremented. Thus, the code following the call to *wait* verifies that the original port remains allocated.

The implementation of *ptsend* enqueues messages in FIFO order. It relies on *pttail* to point to the last node on the queue if the queue is nonempty. Furthermore, *ptsend* always makes *pttail* point to a new node after the node has been added to the list. Finally, *ptsend* signals the receiver semaphore after the new message has been added to the queue, allowing a receiver to consume the message.

As in earlier code, an invariant helps a programmer understand the implementation. The invariant states:

> *Semaphore* ptrsem *has a nonnegative count* n *if* n *messages are waiting in the port; it has negative count* –n *if* n *processes are waiting for messages.*

The call to *panic* also deserves comment because this is its first occurrence. In our design, running out of message nodes is a catastrophe from which the system cannot recover. It means that the arbitrary limit on message nodes, set to prevent ports from using all the free memory, is insufficient. Perhaps the programs using ports are performing incorrectly. Perhaps, through no fault of the user, the system cannot honor a valid request; there is no way to know. Under such circumstances it is often better to announce failure and stop rather than attempt to go on. *Panic* is designed for such situations; it prints the specified error message and halts processing. If the user chooses to continue execution, the call to panic may return, but often the user will restart the system or change the program instead. (The exercises suggest alternative ways of handling the problem.)

## 11.7 Receiving A Message From A Port

Function *ptrecv* implements a basic consumer operation. It removes a message from a specified port and returns the message to its caller. The code is found in file *ptrecv.c*:

```
/* ptrecv.c - ptrecv */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptrecv  --  receive a message from a port, blocking if port empty
```

```
 *------------------------------------------------------------------------
 */
uint32  ptrecv(
          int32         portid          /* ID of port to use            */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  ptentry *ptptr;         /* pointer to table entry       */
        int32   seq;                    /* local copy of sequence num.  */
        umsg32  msg;                    /* message to return            */
        struct  ptnode  *msgnode;       /* first node on message list   */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
                restore(mask);
                return (uint32)SYSERR;
        }

        /* Wait for message and verify that the port is still allocated */

        seq = ptptr->ptseq;             /* record orignal sequence      */
        if (ptptr->ptstate != PT_ALLOC || wait(ptptr->ptrsem) == SYSERR
            || ptptr->ptseq != seq) {
                restore(mask);
                return (uint32)SYSERR;
        }

        /* Dequeue first message that is waiting in the port */

        msgnode = ptptr->pthead;
        msg = msgnode->ptmsg;
        if (ptptr->pthead == ptptr->pttail)     /* delete last item     */
                ptptr->pthead = ptptr->pttail = NULL;
        else
                ptptr->pthead = msgnode->ptnext;
        msgnode->ptnext = ptfree;               /* return to free list  */
        ptfree = msgnode;
        signal(ptptr->ptssem);
        restore(mask);
        return msg;
}
```

*Ptrecv* checks its argument, waits until a message is available, verifies that the port has not been deleted or reused, and dequeues the message node. It records the value of the message in local variable *msg* before returning the message node to the free list, and then returns the value to its caller.

## 11.8 Port Deletion And Reset

It is sometimes useful to delete or to reset a port. In both cases, the system must dispose of waiting messages, return message nodes to the free list, and permit waiting processes to continue execution. How should the port system dispose of waiting messages? It could choose to throw them away, or to return them to the processes that sent them. Often, the user can describe a meaningful disposition, so the design presented allows the user to specify disposition. Functions *ptdelete* and *ptreset* perform port deletion and reset operations. Both take as an argument a function that will be called to dispose of each waiting message. The code is found in files *ptdelete.c* and *ptreset.c*:

```
/* ptdelete.c - ptdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptdelete  --  delete a port, freeing waiting processes and messages
 *------------------------------------------------------------------------
 */
syscall ptdelete(
          int32         portid,         /* ID of port to delete         */
          int32         (*dispose)()    /* function to call to dispose  */
        )                               /*   of waiting messages        */
{
        intmask mask;                   /* saved interrupt mask         */
        struct  ptentry *ptptr;         /* pointer to port table entry  */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
                restore(mask);
                return(SYSERR);
        }
        _ptclear(ptptr, PT_FREE, dispose);
        ptnextid = portid;
        restore(mask);
        return(OK);
}
```

```
/* ptreset.c - ptreset */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptreset  --  reset a port, freeing waiting processes and messages and
                        leaving the port ready for further use
 *------------------------------------------------------------------------
 */
syscall ptreset(
          int32         portid,         /* ID of port to reset          */
          int32         (*dispose)()    /* function to call to dispose  */
                                        /*   of waiting messages        */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  ptentry *ptptr;         /* pointer to port table entry  */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
                restore(mask);
                return SYSERR;
        }
        _ptclear(ptptr, PT_ALLOC, dispose);
        restore(mask);
        return OK;
}
```

Both *ptdelete* and *ptreset* verify that their arguments are correct, and then call *_ptclear* to perform the work of clearing messages and waiting processes.† While it proceeds to clear a port, *_ptclear* places the port in a "limbo" state (*PT_LIMBO*). The limbo state guarantees that no other processes can use the port — functions *ptsend* and *ptrecv* will refuse to operate on a port that is not allocated, and *pcreate* will not allocate the port unless it is free. Thus, *_ptclear* can allow interrupts to remain enabled while it clears a port.

Before declaring a port eligible for use again, *_ptclear* calls *dispose* repeatedly, passing it each waiting message. Finally, after all messages have been removed, *_ptclear* deletes or resets the semaphores as specified by its second argument. Before disposing of messages, *_ptclear* increments the port sequence number so that waiting processes can tell that the port has changed when they awaken. The code is found in file *ptclear.c*:

†The name *_ptclear* begins with an underscore to indicate that the function is internal to the system and is not intended to be called by a user.

```
/* ptclear.c - _ptclear */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  _ptclear  --  used by ptdelete and ptreset to clear or reset a port
 *                       (internal function assumes interrupts disabled
 *                        and arguments have been checked for validity)
 *------------------------------------------------------------------------
 */
void    _ptclear(
          struct ptentry *ptptr,        /* table entry to clear         */
          uint16       newstate,        /* new state for port           */
          int32        (*dispose)(int32)/* disposal function to call    */
         )
{
        struct  ptnode  *walk;          /* pointer to walk message list */

        /* Place port in limbo state while waiting processes are freed */

        ptptr->ptstate = PT_LIMBO;

        ptptr->ptseq++;                 /* reset accession number       */
        walk = ptptr->pthead;           /* first item on msg list       */

        if ( walk != NULL ) {           /* if message list nonempty     */

                /* Walk message list and dispose of each message */

                for( ; walk!=NULL ; walk=walk->ptnext) {
                        (*dispose)( walk->ptmsg );
                }

                /* Link entire message list into the free list */

                (ptptr->pttail)->ptnext = ptfree;
                ptfree = ptptr->pthead;
        }

        if (newstate == PT_ALLOC) {
                ptptr->pttail = ptptr->pthead = NULL;
                sreset(ptptr->ptssem, ptptr->ptmaxcnt);
                sreset(ptptr->ptrsem, 0);
        } else {
                sdelete(ptptr->ptssem);
```

```
                sdelete(ptptr->ptrsem);
        }
        ptptr->ptstate = newstate;
        return;
}
```

## 11.9 Perspective

Because it provides a synchronous interface, our port mechanism allows a process to wait for the next message to arrive. A synchronous interface can be powerful — a clever programmer can use the mechanism to coordinate processes (e.g., to implement mutual exclusion). Interestingly, the ability to coordinate processes also introduces a potential problem: deadlock. That is, a set of processes that uses ports to exchange messages can end up with all processes in the set blocked, waiting for a message to arrive with no processes left running to send a message. Therefore, a programmer using ports must be careful to guarantee that such deadlocks cannot occur.

## 11.10 Summary

The chapter introduces a high-level message passing mechanism, called communication ports, that permits processes to exchange messages through rendezvous points. Each port consists of a fixed length queue of messages. Function *ptsend* deposits a message at the tail of the queue, and function *ptrecv* extracts an item from the head of the queue. Processes that attempt to receive from an empty port are blocked until a message arrives; processes that attempt to send to a full port are blocked until space becomes available.

## EXERCISES

**11.1** Consider the primitives *send—receive* and *ptsend—ptrecv*. Is it possible to design a single message passing scheme that encompasses both? Explain.

**11.2** An important distinction is made between statically allocated and dynamically allocated resources. For example, ports are dynamically allocated while inter-process message slots are statically allocated. What is the key problem with dynamic allocation in a multi-process environment?

**11.3** Change the message node allocation scheme so that a semaphore controls nodes on the free list. Have *ptsend* wait for a free node if none exists. What potential problems, if any, does the new scheme introduce?

**11.4** *Panic* is used for conditions like internal inconsistency or potential deadlock. Often the conditions causing a panic are irreproducible, so their cause is difficult to pinpoint. Discuss what you might do to trace the cause of the panic in *ptsend*.

**11.5** As alternatives to the call to *panic* in *ptsend*, consider allocating more nodes or retrying the operation. What are the liabilities of each?

**11.6** Rewrite *ptsend* and *ptrecv* to return a special value when the port is deleted while they are waiting. What is the chief disadvantage of the new mechanism?

**11.7** Modify the routines in previous chapters that allocate, use, and delete objects so they use sequence numbers to detect deletion as the communication port functions do.

**11.8** *Ptsend* and *ptrecv* cannot transmit a message with value equal to *SYSERR* because *ptrecv* cannot distinguish between a message with that value and an error. Redesign the functions to transmit any value.

# NOTES

# Chapter Contents

# 12

# Interrupt Processing

*The joy of music should never be interrupted by a commercial.*

— Leonard Bernstein

## 12.1 Introduction

Previous chapters focus on processor and memory management. The chapters on processor management introduce the concept of concurrent processing, show how processes are created and terminated, and how processes coordinate. The chapters on memory management illustrate low-level mechanisms used to manage dynamic allocation and the release of stack and heap storage.

This chapter begins a discussion of input and output (I/O) facilities. The chapter reviews the concept of an interrupt, and introduces the overall software architecture that an operating system uses to handle interrupts. It describes an interrupt dispatching mechanism that passes control to the appropriate interrupt handler when an interrupt occurs. More important, the chapter explains the complex relationship between interrupts and the operating system abstraction of concurrent processes, and gives general guidelines that interrupt code must follow to provide a correct and safe implementation of concurrent processes in the presence of interrupts. Later chapters continue the discussion by examining how interrupt handlers are designed for specific devices, including a real-time clock device that enables preemptive process scheduling.

## 12.2 The Advantage Of Interrupts

The interrupt mechanism invented for third-generation computer systems provides a powerful facility that separates I/O activities from processing. Many of the services an operating system offers would not be possible without an interrupt facility.

The motivation for an interrupt mechanism is parallel operation. Instead of relying on the CPU to provide complete control over I/O, each individual device contains hardware that can operate independently. The CPU is only required to start or stop a device — once started, a device proceeds to transfer data without further help. Because most I/O proceeds much slower than computation, the CPU can start multiple devices and allow the operations to proceed in parallel. After starting I/O, the CPU can proceed with other computation (i.e., execute another process) until the device interrupts to signal completion. The key idea is:

> *An interrupt mechanism permits the processor and I/O devices to operate in parallel. Although the details differ, the hardware includes a mechanism that automatically "interrupts" normal processing and informs the operating system when a device completes an operation or needs attention.*

## 12.3 Interrupt Dispatching

Hardware in the processor performs three basic steps when an interrupt occurs:

- Immediately changes the state of the processor to prevent further interrupts from occurring while an interrupt is being processed
- Saves sufficient state to allow the processor to resume execution transparently after the interrupt has been handled
- Branches to a predetermined memory location where the operating system has placed code to process the interrupt

Each processor includes details that complicate interrupt processing. For example, when it saves state, the hardware in most systems does not save a complete copy of all processor registers. Instead, the hardware records a few basic values, such as a copy of the *instruction pointer*,† and requires the operating system to save all other registers that will be used during interrupt processing. The operating system is also required to restore the values before returning to normal processing after the interrupt handling is complete.

†The instruction pointer contains the address of the instruction to be executed next; some architectures use the term *program counter*.

## 12.4 Vectored Interrupts

When an interrupt occurs, the operating system must have a way to identify which device caused the interrupt. A variety of hardware mechanisms have been invented that handle device identification. For example, on some hardware, the operating system must use the bus to ask a device to identify itself. On others, the hardware built into the CPU handles the task automatically. After considering other aspects of interrupt handling, we will discuss an example.

How should a device identify itself? The most popular technique is known as a *vectored interrupt*. Each device is assigned a unique integer, 0, 1, 2, and so on. The integer is called an *interrupt vector number* or *interrupt request number* (*IRQ*). When an interrupt occurs, the device specifies its number. The hardware or the operating system uses the interrupt number as an index into an *interrupt vector array* in memory. The operating system loads each location in the interrupt vector array with a pointer to a function that handles the interrupt associated with that vector. Thus, if a device with interrupt vector number *i* interrupts, control branches to:

interrupt_vector[i]

Figure 12.1 illustrates the array of interrupt vector pointers in memory.

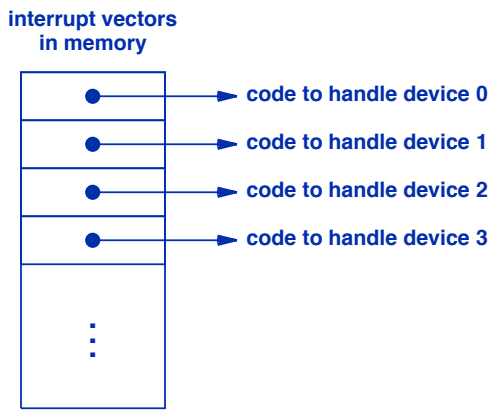



**Figure 12.1** Interrupt vectors in memory, where each entry contains a pointer to the code to handle one device.

## 12.5 Assignment Of Interrupt Vector Numbers

The details of interrupt vector assignment differ widely among computer architectures. Early systems required each device to be assigned a unique interrupt value manually before the device was plugged into a computer (e.g., using switches or wire jumpers on the circuit board). Some systems assigned each device two interrupt vector

numbers: one to be used when the device completed an input operation and the other to be used when the device completed an output operation. Manual assignment had the problems of being tedious and error-prone — if a computer owner accidentally assigned the same interrupt vector number to two different devices, the devices could not operate correctly. Later systems adopted an approach that was less prone to errors. A computer contained sockets into which an I/O device could be plugged, and a unique interrupt vector number was associated with each socket rather than with each device. No matter how the assignment was accomplished, the interrupt address associated with a device had to be coordinated with the operating system because an operating system initializes the interrupt vectors in memory.

Interrupt vector assignments on modern systems are often more dynamic. For example, some devices permit the interrupt number to be *programmable*. When it boots, the operating system uses the bus to determine which I/O devices are present. The system iterates through the set of available devices and chooses a unique interrupt vector number for each device. The system uses the bus to inform the device of its interrupt number, and initializes the interrupt vector in memory to point to the correct handler for the device.

A final approach is used to permit devices to be plugged into a computer while the operating system is running. For example, consider USB devices. The operating system assigns a single hardware interrupt for the USB controller, and configures a device driver for the controller. The driver can load additional driver code for each specific device dynamically. Thus, when a new device is attached, the controller loads the driver for the device and records the driver location. Later, when the device interrupts, the controller receives the interrupt and forwards it to the appropriate driver code.

## 12.6 Interrupt Hardware

Where in memory is the interrupt vector located? What part of interrupt processing does the hardware handle? Approaches vary widely; the details depend on the computer system. Many large computer systems allow an operating system to choose a location for the interrupt vector array. When it boots, the operating system chooses the location, and then informs the hardware of the location by storing the address in an internal hardware register.

On large systems, the hardware handles many of the details without using the CPU. For example, the hardware can use the bus to ask the interrupting device for an interrupt vector number, use the vector number as an index, and branch directly to the interrupt code for the device. On smaller systems, however, the processor must use the bus to determine the interrupt vector number — the processor issues a bus request, and the interrupting device returns its unique interrupt number.

The smallest embedded systems often employ a simplistic interrupt mechanism: a single interrupt location is hardwired, either by being built into the processor chip or the motherboard. Once the interrupt occurs, the operating system must determine which device interrupted, and use the interrupt vector number to jump to the interrupt function

for the specific device. As Chapter 3 describes, the example hardware uses the hardwired approach: whenever a device interrupts, the processor jumps to location 0x80000180. Fortunately, the example system includes a co-processor that handles many details. When an interrupt occurs, the co-processor uses the bus to identify the device. It places a value in the *CAUSE* register to identify the interrupt. Therefore, when it processes an interrupt, the processor does not need to interact directly with the bus hardware or devices. Instead, the processor merely loads the value from the co-processor's *CAUSE* register, and uses the value to determine which device requested the interrupt.

## 12.7 IRQ Limits And Interrupt Multiplexing

Many processors place a limit on the unique interrupt vector numbers that can be assigned (a typical limit is 8). How can a computer have an arbitrary number of devices if the interrupt system limits the vector size? The answer lies in a technique known as *interrupt multiplexing*: a single interrupt number is assigned to multiple devices. When an interrupt occurs, the dispatcher must determine which of the devices assigned the same number needs service.

Interrupt multiplexing works best in situations where multiple devices use a single hardware interface. For example, consider USB devices. From a computer's point of view, a USB hub appears to function as a device connected to the computer's bus. In fact, the USB hub merely provides an additional bus interface to which multiple devices can attach. Whenever a USB device needs service, the device uses the interrupt number associated with the USB, and the operating system passes control to the USB handler. The USB handler can then determine which device needs service and further dispatch control to the code that handles the device.

The Linksys hardware provides another example of interrupt multiplexing. A MIPS processor only permits eight unique interrupt numbers to be assigned, and some are reserved for specific purposes. To meet constraints, the E2100L hardware multiplexes all devices on the system backplane into a single interrupt vector, hardware interrupt *4*. When it receives interrupt *4*, the hardware interrogates the system backplane to determine which device caused the interrupt. For example, if a serial device caused the interrupt, the hardware will identify the interrupt as coming from device *3* on the backplane.

## 12.8 Interrupt Software And Dispatching

Once it has identified the device that caused the interrupt, the operating system calls the function that handles interrupts for the device. We say that the operating system *dispatches* the interrupt to the appropriate *interrupt handler*, and we use the term *dispatcher* to refer to the operating system code that performs the dispatching function. Even on computers in which the hardware can follow the interrupt vector automatically,

many operating systems fill all interrupt vectors with the address of an interrupt dispatcher so the dispatcher can enforce the scheduling invariant and other global system policies. Figure 12.2 illustrates the conceptual division of interrupt processing between a dispatcher and a set of handlers.

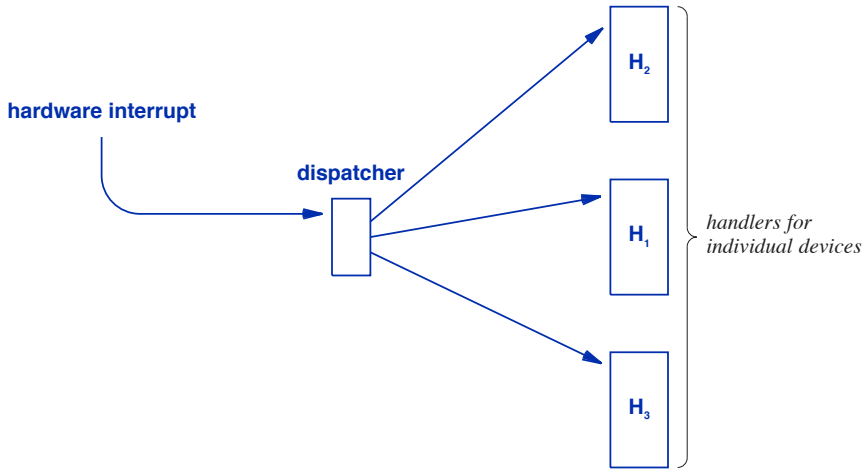



**Figure 12.2** The conceptual organization of interrupt processing software.

In practice, several details lead to a more complex organization. For example, because it interrogates co-processor hardware registers and uses a special instruction to return from an interrupt, some pieces of a MIPS interrupt dispatcher must be written in assembly language. It may seem logical to extend the assembly language code to include all interrupt processing. However, interrupt code accounts for a significant portion of the operating system, and assembly language code is difficult to understand and modify. Therefore, to keep the system code readable, most operating system designers divide a dispatcher into two pieces. The system arranges for a hardware interrupt to branch to a small, low-level piece of code written in assembly language. The low-level code handles tasks such as saving and restoring registers, communicating with the co-processor to identify the interrupting device, and using a special instruction to return from the interrupt once processing is complete. The low-level piece is minimal — as soon as registers have been saved, the low-level piece calls a high-level dispatcher function that is written in C. The high-level dispatcher can examine the interrupt vector array or use other operating system data structures to choose a handler for the interrupting device. Once the address of the handler has been computed, the high-level dispatcher calls the handler function. Despite being divided into two pieces, a dispatcher is small — all the code that communicates with a given device is placed in an *interrupt handler* rather than in the dispatcher itself.

Our example system implements one additional detail. To understand the structure, recall that the example interrupt hardware always transfers to location 0x80000180, and the operating system is loaded at location 0x80001000. Thus, when it boots, the operating system must place code at the reserved location, 0x80000180. Rather than copy the entire low-level interrupt handler to the reserved location, our system stores a single jump instruction at the location which causes the processor to jump to the low-level interrupt dispatcher (label *intdispatch* in the code) whenever an interrupt occurs. Figure 12.3 illustrates the code layout after the operating system has finished initialization.

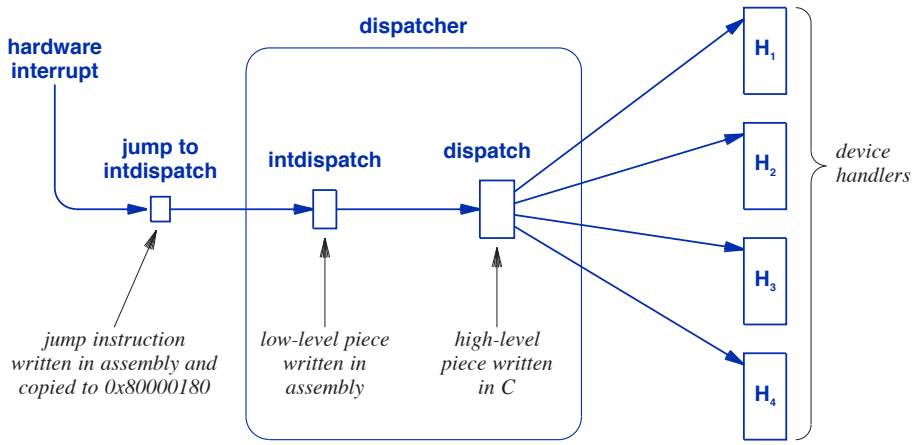



**Figure 12.3** The organization of interrupt code used in the example system.

When it begins, *intdispatch* allocates space on the current run-time stack, and saves each of the processor registers so they can be restored before the interrupt returns. Because the high-level function, *dispatch*, is written in C, *intdispatch* must use C calling conventions. Therefore, after saving registers, *intdispatch* extracts the value from the co-processor *CAUSE* register to determine the cause of the interrupt, and pushes the value on the stack as an argument to *dispatch*. *Intdispatch* also pushes the address of the saved stack frame as a second argument.

The four interrupt handlers depicted in the figure might correspond to the major device types found on the example system:

- Serial line device (console)
- Wired network device (Ethernet)
- Wireless network (Wi-Fi)
- Real-time clock device (timer)

## 12.9 The Lowest Level Of The Interrupt Dispatcher

An examination of the example code will clarify many of the details. File *intdispatch.S* contains the code for the low-level piece of the dispatcher:

```
/* intdispatch.S - intdispatch */

#include <mips.h>
#include <interrupt.h>

.text
        .align 4
        .globl intdispatch

/*------------------------------------------------------------------------
 * intdispatch - low-level piece of interrupt dispatcher
 *------------------------------------------------------------------------
 */
        .ent intdispatch
intdispatch:
        .set noreorder
        .set noat
        j       savestate               /* Jump to low-level handler    */
        nop
savestate:
        addiu   sp, sp, -IRQREC_SIZE    /* Allocate space on stack      */
        sw      AT, IRQREC_AT(sp)       /* Save assembler temp reg first*/
        mfc0    k0, CP0_CAUSE           /* Save interrupt CAUSE value   */
        mfc0    k1, CP0_EPC             /* Save interrupted PC value    */
        sw      k0, IRQREC_CAUSE(sp)
        mfc0    k0, CP0_STATUS          /* Save co-processor STATUS     */
        sw      k1, IRQREC_EPC(sp)
        sw      k0, IRQREC_STATUS(sp)
        .set at
        .set reorder
        sw      v0, IRQREC_V0(sp)       /* Save all general purpose regs*/
        sw      v1, IRQREC_V1(sp)
        sw      a0, IRQREC_A0(sp)
        sw      a1, IRQREC_A1(sp)
        sw      a2, IRQREC_A2(sp)
        sw      a3, IRQREC_A3(sp)
        sw      t0, IRQREC_T0(sp)
        sw      t1, IRQREC_T1(sp)
        sw      t2, IRQREC_T2(sp)
```

```
        sw      t3, IRQREC_T3(sp)
        sw      t4, IRQREC_T4(sp)
        sw      t5, IRQREC_T5(sp)
        sw      t6, IRQREC_T6(sp)
        sw      t7, IRQREC_T7(sp)
        sw      s0, IRQREC_S0(sp)
        sw      s1, IRQREC_S1(sp)
        sw      s2, IRQREC_S2(sp)
        sw      s3, IRQREC_S3(sp)
        sw      s4, IRQREC_S4(sp)
        sw      s5, IRQREC_S5(sp)
        sw      s6, IRQREC_S6(sp)
        sw      s7, IRQREC_S7(sp)
        sw      t8, IRQREC_T8(sp)
        sw      t9, IRQREC_T9(sp)
        sw      k0, IRQREC_K0(sp)
        sw      k1, IRQREC_K1(sp)
        sw      gp, IRQREC_S8(sp)
        sw      sp, IRQREC_SP(sp)
        sw      fp, IRQREC_S9(sp)
        sw      ra, IRQREC_RA(sp)
        sw      zero, IRQREC_ZER(sp)
        mfhi    t0                      /* Save hi and lo               */
        mflo    t1
        sw      t0, IRQREC_HI(sp)
        sw      t1, IRQREC_LO(sp)

        lw      a0, IRQREC_CAUSE(sp)    /* Pass cause and state info to */
        move    a1, sp                  /*  high-level dispatcher       */
        jal     dispatch

restorestate:                           /* On return from dispatcher    */
        lw      t0, IRQREC_HI(sp)       /*  restore all state           */
        lw      t1, IRQREC_LO(sp)
        mthi    t0
        mtlo    t1
        lw      ra, IRQREC_RA(sp)       /* Restore general purpose regs */
        lw      fp, IRQREC_S9(sp)
        lw      gp, IRQREC_S8(sp)
        lw      t9, IRQREC_T9(sp)
        lw      t8, IRQREC_T8(sp)
        lw      s7, IRQREC_S7(sp)
        lw      s6, IRQREC_S6(sp)
        lw      s5, IRQREC_S5(sp)
        lw      s4, IRQREC_S4(sp)
```

```
        lw      s3, IRQREC_S3(sp)
        lw      s2, IRQREC_S2(sp)
        lw      s1, IRQREC_S1(sp)
        lw      s0, IRQREC_S0(sp)
        lw      t7, IRQREC_T7(sp)
        lw      t6, IRQREC_T6(sp)
        lw      t5, IRQREC_T5(sp)
        lw      t4, IRQREC_T4(sp)
        lw      t3, IRQREC_T3(sp)
        lw      t2, IRQREC_T2(sp)
        lw      t1, IRQREC_T1(sp)
        lw      t0, IRQREC_T0(sp)
        lw      a3, IRQREC_A3(sp)
        lw      a2, IRQREC_A2(sp)
        lw      a1, IRQREC_A1(sp)
        lw      a0, IRQREC_A0(sp)
        lw      v1, IRQREC_V1(sp)
        lw      v0, IRQREC_V0(sp)

        .set noreorder
        .set noat
        lw      k0, IRQREC_EPC(sp)      /* Restore interrupted PC value */
        lw      AT, IRQREC_AT(sp)       /* Restore assembler temp reg   */
        mtc0    k0, CP0_EPC
        lw      k1, IRQREC_STATUS(sp)   /* Restore global status reg    */
        addiu   sp, sp, IRQREC_SIZE     /* Restore stack pointer        */
        mtc0    k1, CP0_STATUS
        nop                             /* Delay for co-processor       */
        eret                            /* Return from interrupt        */
        nop
        nop
        .set at
        .set reorder
        .end intdispatch
```

## 12.10 The High-Level Interrupt Dispatcher

Once it has saved a copy of the processor registers and interrogated the co-processor register to determine which device caused the interrupt, the low-level piece of the dispatcher calls function *dispatch*, passing arguments that specify the device and a pointer to the frame that contains the saved status. *Dispatch* uses the cause argument to identify the appropriate interrupt handler, and then calls the handler. Once the interrupt handler returns, *dispatch* returns to the low-level piece of the dispatcher, which restores processor registers and returns from the interrupt. File *dispatch.c* contains the code.

```
/* dispatch.c */

#include <xinu.h>
#include <mips.h>
#include <ar9130.h>

/* Initialize list of interrupts */

char *interrupts[] = {
    "Software interrupt request 0",
    "Software interrupt request 1",
    "Hardware interrupt request 0, wmac",
    "Hardware interrupt request 1, usb",
    "Hardware interrupt request 2, eth0",
    "Hardware interrupt request 3, eth1",
    "Hardware interrupt request 4, misc",
    "Hardware interrupt request 5, timer",
    "Miscellaneous interrupt request 0, timer",
    "Miscellaneous interrupt request 1, error",
    "Miscellaneous interrupt request 2, gpio",
    "Miscellaneous interrupt request 3, uart",
    "Miscellaneous interrupt request 4, watchdog",
    "Miscellaneous interrupt request 5, perf",
    "Miscellaneous interrupt request 6, reserved",
    "Miscellaneous interrupt request 7, mbox",
};

/*------------------------------------------------------------------------
 * dispatch - high-level piece of interrupt dispatcher
 *------------------------------------------------------------------------
 */

void    dispatch(
          int32 cause,          /* identifies interrupt cause           */
          int32 *frame          /* pointer to interrupt frame that      */
                                /*  contains saved status               */
        )
{
        intmask mask;           /* saved interrupt status               */
        int32   irqcode = 0;    /* code for interrupt                   */
        int32   irqnum = -1;    /* interrupt number                     */
        void    (*handler)(void);/* address of handler function to call */

        if (cause & CAUSE_EXC) exception(cause, frame);
```

```
        /* Obtain the IRQ code */

        irqcode = (cause & CAUSE_IRQ) >> CAUSE_IRQ_SHIFT;

        /* Calculate the interrupt number */

        while (irqcode) {
                irqnum++;
                irqcode = irqcode >> 1;
        }

        if (IRQ_ATH_MISC == irqnum) {
                uint32 *miscStat = (uint32 *)RST_MISC_INTERRUPT_STATUS;
                irqcode = *miscStat & RST_MISC_IRQ_MASK;
                irqnum = 7;
                while (irqcode) {
                        irqnum++;
                        irqcode = irqcode >> 1;
                }
        }

        /* Check for registered interrupt handler */

        if ((handler = interruptVector[irqnum]) == NULL) {
                kprintf("Xinu Interrupt %d uncaught, %s\r\n",
                        irqnum, interrupts[irqnum]);
                while (1) {
                        ;               /* forever */
                }
        }

        mask = disable();       /* Disable interrupts for duration */

        exlreset();             /* Reset system-wide exception bit */

        (*handler) ();          /* Invoke device-specific handler  */

        exlset();               /* Set system-wide exception bit   */
        restore(mask);
}

/*------------------------------------------------------------------------
 * enable_irq - enable a specific IRQ
 *------------------------------------------------------------------------
 */
```

```
void enable_irq(
        intmask irqnumber               /* specific IRQ to enable       */
        )
{
        int32   enable_cpuirq(int);
        int irqmisc;
        uint32 *miscMask = (uint32 *)RST_MISC_INTERRUPT_MASK;
        if (irqnumber >= 8) {
                irqmisc = irqnumber - 8;
                enable_cpuirq(IRQ_ATH_MISC);
                *miscMask |= (1 << irqmisc);
        } else {
                enable_cpuirq(irqnumber);
        }
}
```

Function *enable_irq* can be called to enable a specific interrupt. Recall that the hardware uses interrupt multiplexing so that an interrupt from any device on the backplane raises hardware interrupt *4*. Our operating system uses an interesting technique to store the locations of handlers for backplane devices. Although the hardware only uses eight interrupts (corresponding to locations zero through seven in an interrupt vector array), our code builds a larger interrupt vector in memory and uses locations above *7* for backplane devices. That is, location *8* in the array corresponds to backplane device *0*, location *9* corresponds to backplane device *1*, and so on. When the serial device interrupts, hardware interrupt *4* is raised to indicate that a backplane device caused the interrupt. The system interrogates the backplane hardware and determines that backplane device *3* caused the interrupt. The code then adds eight to the device number, and fetches the handler address from interrupt vector location *11*. It may seem that the dispatcher could maintain a separate array for the backplane devices, but placing the information in a single data structure helps keep the system uniform and allows a single piece of code in the dispatcher to invoke any handler, independent of whether interrupt multiplexing was used.

To summarize:

> *Interrupt multiplexing allows multiple devices to share a single interrupt vector; the dispatcher in an operating system can hide dispatching and use a single mechanism to invoke a handler.*

## 12.11 Disabling Interrupts

Because interrupt functions examine and modify global data structures, such as I/O buffers, the operating system must prevent other processes from executing while an interrupt is being processed. As we have seen, when an interrupt occurs, the hardware disables further interrupts, which means interrupt processing is not interruptable. Interrupts remain disabled when the low-level piece of the interrupt dispatcher calls the high-level piece and when the high-level piece calls a handler. When the high-level interrupt dispatcher returns, control passes back to the low-level interrupt dispatcher, which restores the processor state and uses a special assembly language instruction to return to the place at which processing was originally interrupted. During the final step, when the low-level dispatcher returns, interrupts are enabled again. The point can be summarized:

> *Interrupts are disabled before the dispatcher calls a high-level interrupt handler; the high-level handler keeps interrupts disabled while it changes global data structures.*

Although it may seem obvious, the interrupt policy stated above has subtle consequences. Hardware places strict constraints on the number of instructions that can be executed while interrupts are disabled. If an operating system leaves interrupts disabled arbitrarily long, devices will fail to perform correctly. For example, if a processor does not accept a character from an input device before additional characters arrive, one or more characters can be lost. Therefore, interrupt routines must complete processing as quickly as possible and resume executing the code that had interrupts enabled. More important, interrupts are global — if the handler for one device leaves interrupts disabled, all devices are affected. Thus, when creating interrupt code, a programmer must be aware of the constraints on all devices in the system, and must accommodate the device with the smallest time constraint.

> *The maximum time that a handler for device* D *can leave interrupts disabled cannot be computed by examining device* D. *Instead, the time is computed by choosing the smallest constraint across all devices in the system.*

## 12.12 Constraints On Functions That Interrupt Code Invokes

In addition to insuring that interrupt code accommodates the device with the tightest time constraints, an operating system designer must build interrupt code to be executed by an arbitrary process. That is, interrupt code is executed by whichever process is running when the interrupt occurs.

The process seems irrelevant until one considers two facts:

- An interrupt handler can invoke operating system functions.
- Because the scheduler assumes at least one process will remain ready to run, the null process must remain in the *current* or *ready* states.

The null process is designed to be an infinite loop that does not make function calls. However, an interrupt can be thought of as occurring "between" two successive instructions. Thus, if an interrupt occurs while the null process is executing, the null process will remain running while the handler executes. The most important consequence is:

*Interrupt routines can only call operating system functions that leave the executing process in the current or ready states.*

Thus, interrupt routines may invoke functions such as *send* or *signal*, but may not invoke functions such as *wait* that move the executing process to a non-eligible state.

## 12.13 The Need To Reschedule During An Interrupt

Consider the question of rescheduling during an interrupt. To see why rescheduling is needed, observe the following:

- The scheduling invariant specifies that at any time, a highest priority eligible process must be executing.
- When an I/O operation completes, a high-priority process may become eligible to use the processor.

For example, suppose a high-priority process, $P$, chooses to read a packet from the network. Even though it has high priority, $P$ must block to wait for a packet. While $P$ is blocked, some other process, $Q$, will be running when a packet arrives and an interrupt occurs. If the interrupt handler merely moves $P$ to the ready state and returns, process $Q$ will continue to execute. If $Q$ has lower-priority than $P$, the scheduling invariant will be violated.

As an extreme case, consider what happens if a system only contains one application process, and the application blocks to wait for I/O. The null process will be running when the interrupt occurs. If the interrupt handler does not reschedule, the interrupt will return to the null process and the application will never execute. The key idea is:

*To insure that processes are notified promptly when an I/O operation completes and to maintain the scheduling invariant, an interrupt handler must reschedule whenever it makes a waiting process ready.*

## 12.14 Rescheduling During An Interrupt

The interaction between the scheduling and interrupt policies creates a complex question. We said that interrupt routines must keep interrupts disabled while processing an interrupt. We also said that an interrupt handler must re-establish the scheduling invariant whenever a process becomes ready. However, consider what can happen during rescheduling. Suppose the process that is selected to execute has been executing with interrupts enabled. Once it begins to execute, the process will enable interrupts. Thus, it might seem that an interrupt handler should not be allowed to reschedule because switching to a process that has interrupts enabled could start a cascade of further interrupts. We must convince ourselves that rescheduling during an interrupt is safe as long as global data structures are valid.

To understand why rescheduling is safe, consider the series of events leading to a call of *resched* from an interrupt handler. Suppose a process *U* was running with interrupts enabled when the interrupt occurred. The low-level interrupt dispatch code uses *U*'s stack to save the state, and leaves process *U* running with interrupts disabled while the high-level dispatcher executes. Interrupts remained disabled when the high-level dispatcher calls the interrupt handler. Suppose that during the sequence, the code calls *resched*, which switches to another process, *T*. If *T* happens to enable interrupts (e.g., by returning from a system call), another interrupt may occur. What prevents an infinite loop where unfinished interrupts pile up until a stack overflows with interrupt procedure calls? Recall that each process has its own stack. Process *U* had one interrupt on its stack when it was stopped by the context switch. The new interrupt occurs while the processor is using *T*'s stack. Before another interrupt can pile up on *U*'s stack, it must regain control of the CPU and enable interrupts. Recall, *U* was running with interrupts disabled when it called the scheduler and context switch. The context switch saved *U* with interrupts disabled, so when it switches back to *U*, the context switch code will restore the interrupt status, and *U* will continue execution with interrupts disabled.

Interrupts remain disabled as *resched* returns to the interrupt handler and as the interrupt handler returns to the dispatcher. Interrupts only become enabled again when the dispatcher returns to the location at which the original interrupt occurred. So, no additional interrupts can occur while process *U* is executing interrupt code (even though an interrupt can occur if *U* switches to another process and the other process runs). That is, only one interrupt can be in progress for a given process at any time. Because only a finite number of processes exist in the system at a given time and each process can have at most one outstanding interrupt, the number of outstanding interrupts is bounded. The key point is:

*Rescheduling during interrupt processing is safe provided that (1) interrupt routines leave global data in a valid state before rescheduling, and (2) no function enables interrupts unless it disabled them.*

The rule explains why all operating systems functions use disable/restore rather than disable/enable. A function that disables interrupts upon entry always restores them before returning to its caller; no routine ever enables interrupts explicitly. Because interrupts are disabled upon entry to the interrupt dispatcher, they are restored when it returns. The only exception to our rule about disabling and restoring interrupts is found in the initialization function which enables interrupts at system startup.

## 12.15 Perspective

The relationship between interrupts and processes is among the most subtle and complex aspects of operating systems. Interrupts are low-level mechanisms — they are part of the underlying hardware and are defined in terms of sequential notions such as the fetch-execute cycle. Processes are high-level abstractions — they are imagined by operating system designers and defined by a set of system functions. Consequently, it is easiest to think of interrupts without thinking about concurrent processes and easiest to understand concurrent processes without thinking about interrupts.

Unfortunately, combining the abstract world of processes and the concrete world of interrupts is intellectually challenging. If the interactions between interrupts and processes does not seem incredibly complex, you have not thought about it deeply. If it seems too complex to grasp, console yourself that you are not alone, and with careful thought you will be able to master the basics.

## 12.16 Summary

To process an interrupt, the operating system saves a copy of the processor state, determines which device requested the interrupt, and invokes a handler for the device. Because a high-level language such as C does not provide facilities to manipulate processor or co-processor registers directly, some interrupt processing code cannot be written in C. Rather than write all interrupt code in assembly language, the example system divides the interrupt dispatcher into two pieces: a small, low-level piece written in assembly language and a high-level piece written in C.

A dispatcher catches interrupts, saves machine registers, determines which device requested the interrupt, and passes control to an appropriate high-level interrupt handler. When the high-level handler returns, control passes back to the dispatcher which reloads registers and executes special instructions that restore the state and return to the interrupted program.

Several rules control interrupt processing. First, interrupt code must not leave interrupts disabled arbitrarily long or devices will fail to operate correctly. The length of time an interrupt can be delayed depends on all devices attached to the system, not only on the device being serviced. Second, because it can be executed by the null process, interrupt code must never call a function that will move the executing process out of the *current* or *ready* states. Third, an interrupt handler must not enable interrupts explicitly.

Despite the prohibition on enabling interrupts, a handler must reschedule whenever a waiting process becomes ready. Doing so re-establishes the scheduling invariant and also means that if a process is waiting for I/O to complete, the process will be informed promptly. Of course, the code must insure that global data structures are in a valid state before rescheduling. Rescheduling does not cause a cascade of interrupts because each process can have at most one interrupt on its stack.

## EXERCISES

**12.1** Suppose an interrupt handler contains an error that explicitly enables interrupts. Describe how a system might fail.

**12.2** Modify interrupt handlers to enable interrupts, and see how long a system can run before crashing. Are you surprised? Determine *exactly* why the system crashes. (Note: for this exercise, disable the timer device that is described in the next chapter.)

**12.3** Imagine a processor where the hardware automatically switches context to a special "interrupt process" whenever an interrupt occurs. The only purpose of the interrupt process is to run interrupt code. Does such a design make an operating system easier or more difficult to design? Explain. Hint: will the interrupted process be permitted to reschedule?

**12.4** As the chapter points out, some computer systems allow the system to assign a separate interrupt location to each device and arrange for the hardware to pass control directly to the hander. Thus, no dispatching software is needed. What is the chief disadvantage of such a scheme?

**12.5** Calculate how many microseconds can be spent per interrupt assuming eight serial devices each receive characters at 115 Kbaud (115 thousand bits per second, or approximately 11,500 characters per second).

# NOTES

# Chapter Contents

# 13

# Real-Time Clock Management

*We haven't the time to take our time.*

— Eugene Ionesco

## 13.1 Introduction

Earlier chapters describe two major pieces of an operating system: a processor management system that provides concurrent processing and a memory manager that allows blocks of memory to be allocated and released dynamically. The previous chapter introduces interrupt processing. The chapter states rules for interrupt processing, describes how the operating system captures control when an interrupt occurs, and explains how control passes through a dispatcher to a device-specific interrupt handler.

This chapter continues the discussion of interrupts by describing timing hardware and explaining how an operating system uses a real-time mechanism to provide processes with the ability to control timed events. The chapter introduces two fundamental concepts: a delta list data structure and process preemption. It explains how an operating system uses a clock to provide round-robin service to a set of equal-priority processes. Later chapters extend the study of interrupts by exploring other I/O devices.

## 13.2 Timed Events

Many applications use *timed events*. For example, an application might create a window to display a message, leave the window on the screen for five seconds, and then remove the window. An application that prompts for a password might choose to exit unless a password is entered within thirty seconds. Parts of an operating system also use timed events. For example, many network protocols require a sender to retransmit a request if no response is received within a specified time. Similarly, an operating system might choose to inform a user if a peripheral, such as a printer, remains disconnected for more than a few seconds. On small embedded systems that do not have a separate hardware mechanism, an operating system uses timed events to maintain the current date and time of day.

Because time is fundamental, most operating systems provide facilities that make it easy for an application to create and manage a set of timed events. Some systems use a general-purpose *asynchronous event paradigm* in which a programmer defines a set of event handlers and the operating system invokes the appropriate handler when an event occurs. Timed events fit into the asynchronous paradigm easily: a running process requests that a specific event occur $T$ time units in the future. Other systems follow a *synchronous event paradigm* in which the operating system only provides delay and a programmer creates extra processes as needed to schedule events. Our example system uses the synchronous approach.

## 13.3 Real-Time Clocks And Timer Hardware

Four types of hardware devices relate to time:

- Processor clock
- Real-time clock
- Time-of-day clock
- Interval timer

*Processor clock.* The term *processor clock* refers to a hardware device that emits pulses (i.e., square waves) at regular intervals with high precision. The processor clock controls the rate at which the processor executes instructions. To minimize hardware, low-end embedded systems often use the processor clock as a source of timing information. Unfortunately, a processor clock rate is often inconvenient (i.e., the clock pulses rapidly, and the rate is not a power of ten).

*Real-time clock.* A real-time clock operates independent of the CPU, and pulses in fractions of a second (e.g., 1000 times per second), generating an interrupt for each pulse. Usually real-time clock hardware does not count pulses — if an operating system needs to compute an elapsed time, the system must increment a counter when each clock interrupt occurs.

*Time-of-day clock*. Technically, a time-of-day clock is a chronometer that computes elapsed time. The hardware comprises an internal real-time clock and a counter that tallies the pulses. Like a normal clock, the time can be changed; once set, however, the mechanism runs independent of the processor, and continues as long as the system receives power (some units include a small battery to keep the clock active even if the external power is removed). Unlike other clocks, a time-of-day clock does not interrupt — the processor must set or interrogate the counter.

*Interval timer*. An interval timer, sometimes called a *count-down timer* or simply a *timer*, consists of an internal real-time clock and a counter. To use a *timer*, the system initializes the counter to a positive value. The timer decrements the count once for each real-time clock pulse, and generates an interrupt when the count reaches zero. A variant known as a *count-up timer* requires the operating system to initialize the count to zero and set a *limit*. As the name implies, a count-up timer increments the counter, and interrupts the operating system when the counter reaches the limit value.

The chief advantage of a timer over a real-time clock lies in lower interrupt overhead. A real-time clock interrupts regularly, even if the next event is many time units in the future. A timer only interrupts when an event is scheduled. Furthermore, a timer is more flexible than a real-time clock because a timer can emulate a real-time clock. To emulate a real-time clock with a rate of $R$ pulses per second, for example, a timer is set to interrupt in $1/R$ seconds. When an interrupt occurs, the timer is reset to the same value. To summarize:

> *The hardware available for timed events consists of real-time clocks and interval timers. An operating system can use a timer to emulate a real-time clock by computing a time between pulses, T, and resetting the timer to T on each interrupt.*

The E2100L hardware includes an interval timer as described above. The example code, described later in the chapter, illustrates that using the timer to emulate a real-time clock is straightforward.

## 13.4 Handling Real-Time Clock Interrupts

We said that a real-time clock interrupts once per pulse without counting or accumulating interrupts. Similarly, if a timer is used to emulate a real-time clock, the timer does not accumulate interrupts. In either case, if a processor fails to service a clock interrupt before the clock pulses again, the processor will not receive the second interrupt. More important, the hardware does not detect or report the error:

> *If a processor takes too long to service a real-time clock interrupt or if it operates with interrupts disabled for more than one clock cycle, a clock interrupt will be missed and no error will be reported.*

The operation of real-time clock hardware has two significant consequences for system designers. First, because it must be able to execute many instructions between real-time clock interrupts, a processor must operate significantly faster than the real-time clock. Second, real-time clock interrupts can be a source of hidden errors. That is, if an operating system runs too long with interrupts disabled, clock interrupts will be missed and timing will be affected. Such errors can easily go undetected.

Obviously, systems must be designed to service clock interrupts quickly. Some hardware helps by giving highest priority to real-time clock interrupts. Thus, if an I/O device and a clock device each request an interrupt at the same time, the processor receives the clock interrupt first, and only receives the I/O interrupt after the clock has been serviced.

## 13.5 Delay And Preemption

We will focus on two ways that an operating system uses time:

- Timed delay
- Preemption

*Timed delay*. An operating system allows any process to request a timed delay. When a process requests a timed delay, the operating system moves the process from the current state into a new state (which we call *sleeping*), and schedules a *wakeup* event to restart the process at the specified time. When the wakeup event occurs, the process becomes eligible to use the processor, and executes according to the scheduling policy. Later sections explain how a process is put to sleep and how it is reawakened at the correct time.

*Preemption*. The process manager in an operating system uses a preemption mechanism to implement *time slicing* that guarantees equal-priority processes receive service round-robin, as specified by the scheduling policy in Chapter 5. The system defines a maximum time slice, $T$, that a process can execute without allowing other processes to execute. When it switches from one process to another, the scheduler schedules a preemption event $T$ time units in the future. When a preemption event occurs, the event handler simply calls *resched*.

To understand how preemption works, observe that a system may contain multiple processes with the same priority. Thus, while one process executes, other processes of equal priority may be enqueued on the ready list, eligible to run. In such cases, a call to *resched* places the current process at the end of the ready list, behind other processes with equal priority, and switches to the first process on the list. Therefore, if $k$ equal-priority processes are ready to use the processor, all $k$ execute for at most one time slice before any process receives more service.

How long should a time slice be? We say that the choice of a time slice controls the *granularity of preemption*. Using a short time slice makes the granularity small by rescheduling often. Small granularity tends to keep all equal priority processes proceed-

ing at approximately the same pace. However, a small granularity introduces higher overhead because the system switches context often. A large granularity reduces the overhead of context switching, but allows a process to hold the processor longer before allowing other processes to execute.

It turns out that in most systems, a process seldom uses the processor long enough for preemption to occur. Instead, a process usually performs I/O or executes a system function, such as *wait*, that causes rescheduling. In essence, a process voluntarily gives up control of the processor before the timeslice ends. More important, because input and output are slow compared to processing, processes spend most of their time waiting for I/O to complete. Despite the expected case, preemption provides important functionality:

> *Without a preemptive capability, an operating system cannot regain control from a process that executes an infinite loop.*

## 13.6 Emulating A Real-Time Clock With A Timer

We use the term *clock tick* to refer to a real-time clock interrupt, and the term *tick rate* to refer to the rate at which a clock ticks. Because it uses timer hardware to emulate a real-time clock, our example system has the ability to choose a convenient tick rate. We have chosen to make the clock tick every millisecond. Thus, when a timer interrupt occurs, the system resets the timer delay to a millisecond (i.e., one-thousandth of a second), which will cause another interrupt a millisecond later.

Unfortunately, the timer hardware on the E2100L uses the same technique as many small embedded devices: the hardware uses the processor clock to increment the counter. That is, the interval timer measures time in processor cycles — if the processor clock has a rate of $P$ cycles per second, the timer will count $P$ times in a second. To count a millisecond, we must set the timer to the number of processor cycles in a millisecond.

Fortunately, the speed of the processor clock in the E2100L hardware is known, making it easy to compute the number of processor cycles that will elapse in one millisecond. In the example code, the value has been precomputed and stored in symbolic constant *CLKCYCS_PER_TICK*. When a timer interrupt occurs, the timer is reset to the previous value plus *CLKCYCS_PER_TICK*, which will cause another interrupt in one millisecond.

## 13.7 Implementation Of Preemption

The example code implements both preemption and timed delays; before examining the code, we will discuss each. Preemption is the easiest to understand. Defined constant *QUANTUM* specifies the number of clock ticks in a single time slice. Whenever it switches from one process to another, *resched* sets global variable *preempt* to

*QUANTUM*. Each time the emulated clock ticks, the clock interrupt handler decrements *preempt*. When *preempt* reaches zero, the clock interrupt handler resets *preempt* to *QUANTUM* and calls *resched*. Following the call to *resched*, the handler returns from the interrupt.

The call to *resched* has two possible outcomes. First, if the currently executing process remains the only process at the highest priority, *resched* will return immediately, the interrupt handler will return, and the current process will continue executing at the point of the interrupt. Second, if another ready process has the same priority as the current process, *resched* will switch to the new process. Eventually, *resched* will switch back to the interrupted process. The assignment of *QUANTUM* to *preempt* handles the case where the current process remains running. The assignment is needed because *resched* only resets the preemption counter when it switches to a new process.† Resetting the preemption counter prevents the counter from underflowing in cases where a single process executes indefinitely.

## 13.8 Efficient Management Of Delay With A Delta List

To implement delay, the operating system must maintain a set of processes that have requested a delay. Each process specifies a delay relative to the time at which it places a request, and additional processes can make a request at any time. When the delay for a process expires, the system makes the process ready and calls *resched*.

How can an operating system maintain a set of processes that have each requested a specific delay? The system cannot afford to search through arbitrarily long lists of sleeping processes on each clock tick. Therefore, an efficient data structure is needed that only requires a clock interrupt handler to execute a few instructions on each clock tick while accommodating a set of processes that have each requested a specific delay.

The solution lies in a data structure that we call a *delta list*. A delta list contains a set of processes, and the list is ordered by the time at which a process should awaken. The fundamental insight needed to make computation efficient lies in the use of *relative* rather than *absolute* times. That is, instead of storing a value that specifies the time a process should awaken, a key in the delta list stores the additional time a process must delay beyond the preceding process:

> *The key of the first process on a delta list specifies the number of clock ticks a process must delay beyond the current time; the key of each other process on a delta list specifies the number of clock ticks the process must delay beyond the preceding process on the list.*

As an example, suppose processes A, B, C, and D request delays of 6, 12, 27, and 50 ticks, respectively. Further suppose the requests are made at approximately the same time (i.e., within one clock tick). Figure 13.1 illustrates the delta list that will result.

†The code for *resched* can be found on page 74.

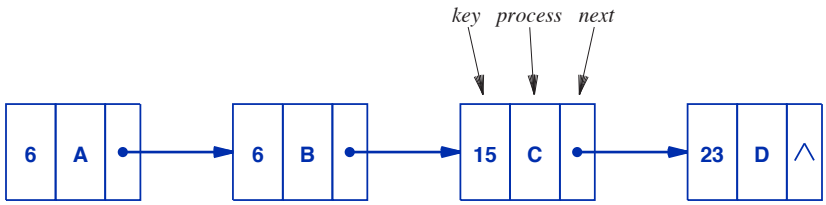



**Figure 13.1** Conceptual form of a delta list with four processes that have delays 6, 12, 27, and 50 ticks, respectively.

Given a delta list, one can find the time at which each process will awaken by computing partial sums of keys. In the figure, the delay before process *A* awakens is 6, the delay before process *B* awakens is 6+6, the delay before process *C* awakens is 6+6+15, and the delay before *D* awakens is 6+6+15+23.

## 13.9 Delta List Implementation

Like other lists of processes, the delta list of delayed processes resides in the *queuetab* structure. Global variable *sleepq* contains the queue ID of the delta list for sleeping processes. On each clock tick, the clock interrupt handler decrements the key on the first item in *sleepq*. If the value reaches zero, the clock handler calls function *wakeup* to awaken the first process because its delay has expired.

Recall that the first process on the sleeping process queue is a process with least delay, and its key specifies the remaining delay in clock ticks until the process must awaken. Because all successive delays in the list are relative to the first delay, the clock interrupt routine only needs to decrement the first key, and does not need to scan the list.

Functions to manipulate a delta list seem straightforward, but the implementation can be tricky. Therefore, a programmer must pay close attention to details. Function *insertd*, shown below, takes three arguments: a process ID, *pid*, a queue ID, *q*, and a delay given by argument *key*. *Insertd* computes a relative delay, and inserts the specified process in *sleepq* at the appropriate location. In the code, variable *next* scans the delta list searching for the place to insert the new process.

Observe that the initial value of argument *key* specifies a delay relative to the current time. Thus, argument *key* can be compared to the key in the first item on the delta list. However, successive keys in the delta list specify delays relative to their predecessor. Thus, the key in successive nodes on the list cannot be compared directly to the value of argument *key*. To keep the delays comparable, *insertd* subtracts the relative delays from *key* as the search proceeds, maintaining the following invariant:

*At any time during the search, both key and queuetab[next].qkey specify a delay relative to the time at which the predecessor of "next" awakens.*

```
/* insertd.c - insertd */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  insertd  -  Insert a process in delta list using delay as the key
 *------------------------------------------------------------------------
 */
status  insertd(                        /* assumes interrupts disabled  */
          pid32         pid,            /* ID of process to insert      */
          qid16         q,              /* ID of queue to use           */
          int32         key             /* delay from "now" (in ms.)    */
        )
{
        int     next;                   /* runs through the delta list  */
        int     prev;                   /* follows next through the list*/

        if (isbadqid(q) || isbadpid(pid)) {
                return SYSERR;
        }

        prev = queuehead(q);
        next = queuetab[queuehead(q)].qnext;
        while ((next != queuetail(q)) && (queuetab[next].qkey <= key)) {
                key -= queuetab[next].qkey;
                prev = next;
                next = queuetab[next].qnext;
        }

        /* Insert new node between prev and next nodes */

        queuetab[pid].qnext = next;
        queuetab[pid].qprev = prev;
        queuetab[pid].qkey = key;
        queuetab[prev].qnext = pid;
        queuetab[next].qprev = pid;
        if (next != queuetail(q)) {
                queuetab[next].qkey -= key;
        }
```

```
        return OK;
}
```

Although *insertd* checks for the tail of the list explicitly during the search, the test could be removed without affecting the execution. To understand why, recall that the key value in the tail of a list is assumed to be greater than any key being inserted. As long as the assertion holds, the loop will terminate once the tail has been reached. Because *insertd* does not check its argument, keeping the test provides a safety check.

After it has identified a location on the list where the relative delay of the item being inserted is smaller than the relative delay of an item on the list, *insertd* links the new item into the list. *Insertd* must also subtract the extra delay that the new item introduces from the delay of the rest of the list. To do so, *insertd* decrements the key in the next item on the list by the key value being inserted. The subtraction is guaranteed to produce a non-negative value because the loop termination condition guarantees that the key inserted is less than the key on the list.

## 13.10 Putting A Process To Sleep

An application does not call *insertd*, nor does the application access the sleep queue directly. Instead, an application invokes system call *sleep* or *sleepms* to request a delay. The only difference between the two functions arises from the granularity of their arguments: an argument to *sleepms* specifies a delay in milliseconds, and an argument to *sleep* specifies a delay in seconds. On a 32-bit processor, measuring delay in milliseconds provides an adequate range of delay. An unsigned 32-bit integer accommodates delays over 1100 hours (49 days). On embedded systems that use 16-bit integers, however, millisecond delays mean that a caller can only express a delay of thirty-two seconds. Thus, an operating system designed for a 16-bit processor usually uses a larger granularity measurement (e.g., tenths of seconds).

To avoid duplicating code, function *sleep* multiplies its argument by 1000 and invokes *sleepms*. The only interesting aspect of *sleep* is a check on its argument size: to avoid integer overflow, *sleep* limits the delay to a value that can be represented as a 32-bit unsigned integer. If the caller specifies a larger value, *sleep* returns *SYSERR*.

*Sleepms* inserts the calling process into the delta list of sleeping processes. When it has been moved to the list of sleeping processes, a process is no longer *ready* or *current*. In what state should it be placed? Sleeping differs from suspension, waiting to receive a message, or waiting for a semaphore. Thus, because none of the existing states suffices, a new process state must be added to the design. We call the new state *sleeping*, and denote it with symbolic constant *PR_SLEEP*. Figure 13.2 illustrates state transitions that include the sleeping state.

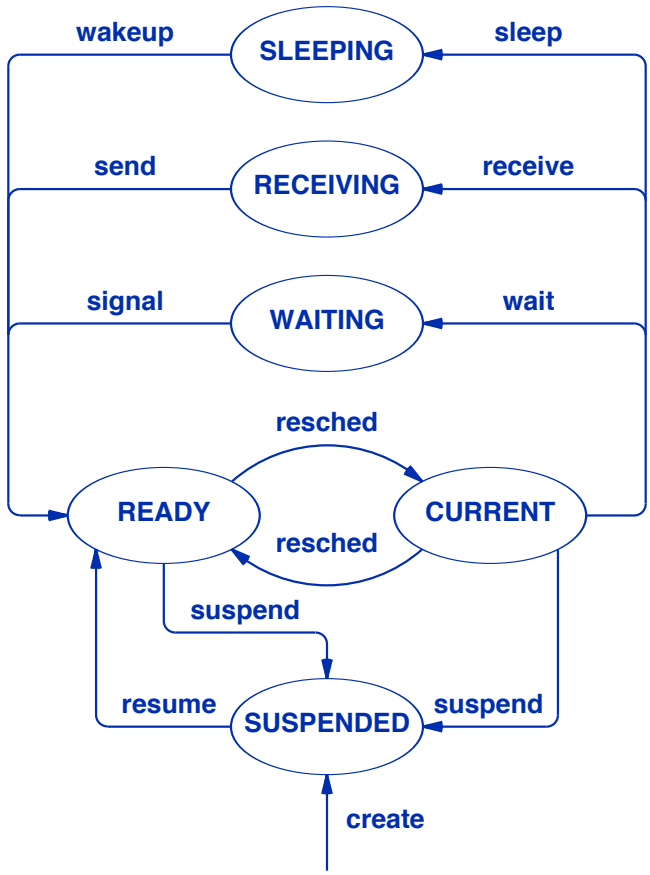



**Figure 13.2** State transitions including the *sleeping* state.

The implementation of *sleepms*, shown below in file *sleep.c*, includes a special case: if a process specified a delay of zero, *sleepms* calls *resched* immediately. Otherwise, *sleepms* uses *insertd* to insert the current process in the delta list of sleeping processes, changes the state to *sleeping*, and calls *resched* to allow other processes to execute.

```
/* sleep.c - sleep sleepms */

#include <xinu.h>

#define MAXSECONDS      4294967         /* max seconds per 32-bit msec  */

/*------------------------------------------------------------------------
```

```
 *  sleep  -  Delay the calling process n seconds
 *------------------------------------------------------------------------
 */
syscall sleep(
          uint32        delay           /* time to delay in seconds     */
        )
{
        if (delay > MAXSECONDS) {
                return(SYSERR);
        }
        sleepms(1000*delay);
        return OK;
}

/*------------------------------------------------------------------------
 *  sleepms  -  Delay the calling process n milliseconds
 *------------------------------------------------------------------------
 */
syscall sleepms(
          uint32        delay           /* time to delay in msec.       */
        )
{
        intmask mask;                   /* saved interrupt mask         */

        mask = disable();
        if (delay == 0) {
                yield();
                restore(mask);
                return OK;
        }

        /* Delay calling process */

        if (insertd(currpid, sleepq, delay) == SYSERR) {
                restore(mask);
                return SYSERR;
        }
        proctab[currpid].prstate = PR_SLEEP;
        resched();
        restore(mask);
        return OK;
}
```

## 13.11 Timed Message Reception

Xinu includes a mechanism that is especially useful in computer networking: timed message reception. In essence, the mechanism allows a process to wait for a specified time or for a message to arrive, whichever occurs first. That is, the mechanism operates like the synchronous *receive* function with an additional provision that places a bound on the maximum time the process will wait.

The fundamental concept behind timed message reception is *disjunctive wait*: a process blocks until one of two events occurs. Many network protocols use disjunctive wait to implement timeout-and-retransmission, a technique senders employ to handle packet loss. When it sends a message, a sender also starts a timer, and then waits for a reply to arrive or the timer to expire, whichever occurs first. If a reply arrives, the network cancels the timer. If the message or the reply is lost, the timer expires, and the protocol software retransmits a copy of the request.

In Xinu, when a process requests a timed receive, the process is placed on the queue of sleeping processes, exactly like any other sleeping process. Instead of assigning the process state *PR_SLEEP*, however, the system places the process in state *PR_RECTIM* to indicate that it is engaged in a receive with timeout. If the sleep timer expires, the process is awakened like any other sleeping process. If a message arrives before the delay expires, *send* calls *unsleep* to remove the process from the queue of sleeping processes, and proceeds to deliver the message. Once it resumes execution, the process checks its process table entry to see if a message has arrived. If no message is present, the timer must have expired.

Figure 13.3 shows the state diagram with a new state, *TIMED-RECV*, for timed message reception.

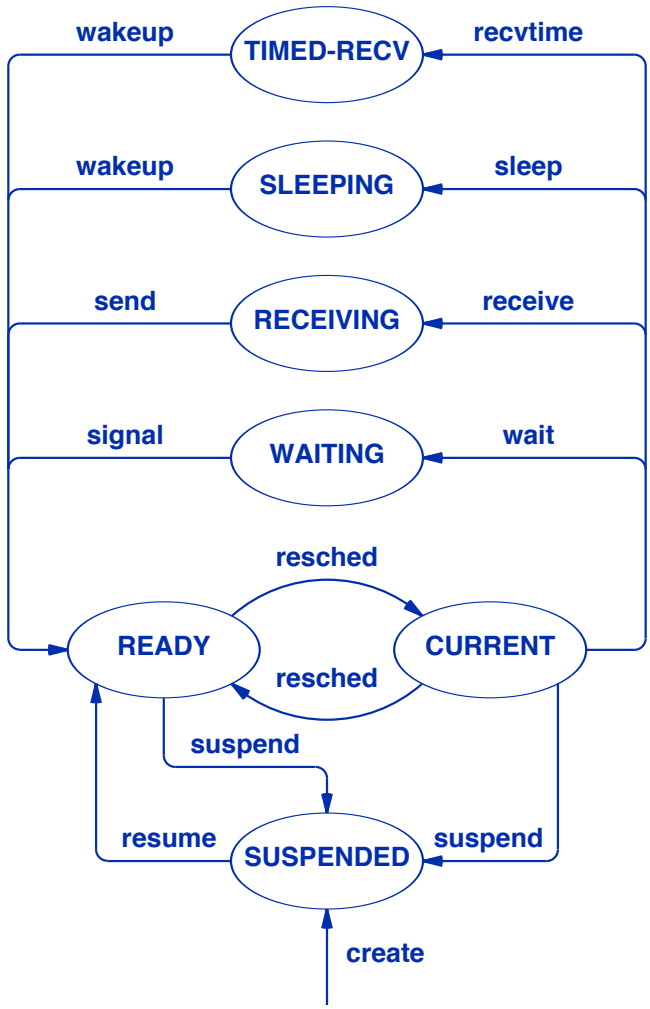



**Figure 13.3** State transitions including the *timed receive* state.

As we have seen, the *send*† function in Chapter 8 handles the case where a process is in the timed receive state. Thus, we only need to examine the code for function *recvtime* and *unsleep*. Function *recvtime* is almost identical to function *receive*‡ except that before calling *resched*, *recvtime* calls *insertd* to insert the calling process on the queue of sleeping processes and assigns state *PR_RECTIM* instead of state *PR_RECV*. File *recvtime.c* contains the code.

†Function *send* can be found in file *send.c* on page 133.
‡Function *receive* can be found in file *receive.c* on page 134.

```
/* recvtime.c - recvtime */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  recvtime  -  wait specified time to receive a message and return
 *------------------------------------------------------------------------
 */
umsg32  recvtime(
          int32         maxwait         /* ticks to wait before timeout */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* tbl entry of current process */
        umsg32  msg;                    /* message to return            */

        if (maxwait < 0) {
                return SYSERR;
        }
        mask = disable();

        /* Schedule wakeup and place process in timed-receive state */

        prptr = &proctab[currpid];
        if (prptr->prhasmsg == FALSE) { /* if message waiting, no delay */
                if (insertd(currpid,sleepq,maxwait) == SYSERR) {
                        restore(mask);
                        return SYSERR;
                }
                prptr->prstate = PR_RECTIM;
                resched();
        }

        /* Either message arrived or timer expired */

        if (prptr->prhasmsg) {
                msg = prptr->prmsg;     /* retrieve message             */
                prptr->prhasmsg = FALSE;/* reset message indicator      */
        } else {
                msg = TIMEOUT;
        }
        restore(mask);
        return msg;
}
```

Function *unsleep* removes a process from the queue of sleeping processes. If other processes are present on the queue, *unsleep* must adjust the delay on subsequent processes to compensate for the process being removed (i.e., must add the process's delay to the next process). File *unsleep.c* contains the code.

```
/* unsleep.c - unsleep */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  unsleep  -  Remove a process from the sleep queue prematurely by
 *                      adjusting the delay of successive processes
 *------------------------------------------------------------------------
 */
syscall unsleep(
          pid32         pid             /* ID of process to remove      */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */

        pid32   pidnext;                /* ID of process on sleep queue */
                                        /* that follows the process that*/
                                        /* is being removed             */

        mask = disable();

        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }

        /* Verify that candidate process is on the sleep queue */

        prptr = &proctab[pid];
        if ((prptr->prstate!=PR_SLEEP) && (prptr->prstate!=PR_RECTIM)) {
                restore(mask);
                return SYSERR;
        }

        /* Increment delay of next process if such a process exists */

        pidnext = queuetab[pid].qnext;
        if (pidnext < NPROC) {
```

```
                queuetab[pidnext].qkey += queuetab[pid].qkey;
        }

        getitem(pid);                   /* unlink process from queue */
        restore(mask);
        return OK;
}
```

## 13.12 Awakening Sleeping Processes

The clock interrupt handler decrements the count of the first key on *sleepq* on each clock tick, calling *wakeup* to awaken the process when the delay reaches zero. *Wakeup* must handle the case where multiple processes are scheduled to awaken at the same time. Thus, *wakeup* iterates through all processes that have a key of zero, removing the process from *sleepq* and calling *ready* to make the process eligible for CPU service again. Because it has been called from an interrupt dispatcher, *wakeup* assumes that interrupts have been disabled upon entry. Thus, *wakeup* does not explicitly disable interrupts before calling *ready*. Once it finishes moving processes to the ready list, *wakeup* calls *resched* to re-establish the scheduling invariant and allow another process to execute.

```
/* wakeup.c - wakeup */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  wakeup  -  Called by clock interrupt handler to awaken processes
 *------------------------------------------------------------------------
 */
void    wakeup(void)
{
        /* Awaken all processes that have no more time to sleep */

        while (nonempty(sleepq) && (firstkey(sleepq) <= 0)) {
                ready(dequeue(sleepq), RESCHED_NO);
        }
        resched();
        return;
}
```

## 13.13 Clock Interrupt Processing

We are now ready to examine the clock interrupt handler, *clkhandler*, which is called each time the emulated clock interrupts. As described above, the clock interrupt handler decrements the time remaining on the first process in *sleepq* (provided *sleepq* is nonempty). If the remaining delay reaches zero, *clkint* calls *wakeup* to remove all processes with zero delay from the sleep queue and make them *ready*. Finally, *clkint* decrements the preemption counter, calling *resched* if the preemption counter reaches zero.

```
/* clkhandler.c - clkhandler */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  clkhandler - handle clock interrupt and process preemption events
 *                       as well as awakening sleeping processes
 *------------------------------------------------------------------------
 */
interrupt clkhandler(void)
{
        clkupdate(CLKCYCS_PER_TICK);

        /* record clock ticks */

        clkticks++;

        /* update global counter for seconds */

        if (clkticks == CLKTICKS_PER_SEC) {
                clktime++;
                clkticks = 0;
        }

        /* If sleep queue is nonempty, decrement first key; when the    */
        /* key reaches zero, awaken a sleeping process                  */

        if (nonempty(sleepq) && (--firstkey(sleepq) <= 0)) {
                wakeup();
        }

        /* Check to see if this proc should be preempted */

        if (--preempt <= 0) {
                preempt = QUANTUM;
                resched();
```

```
        }
        return;
}
```

## 13.14 Clock Initialization

The clock initialization routing, *clkinit*, performs four main functions. First, it allocates a queue to hold the delta list of sleeping processes, and stores the queue ID in global variable *sleepq*. Second, it starts the one-second timer at zero. Third, the code stores the address of the clock interrupt handler in array *interruptVector*, which allows the interrupt dispatcher to associate *clkhandler* with a timer interrupt. Finally, *clkinit* calls *clkupdate* to update the interval timer. The initialization code can be found in file *clkinit.c*.

```
/* clkinit.c */

#include <xinu.h>
#include <interrupt.h>
#include <clock.h>

uint32  clkticks = 0;           /* ticks per second                     */
uint32  clktime = 0;            /* current time in seconds              */
qid16   sleepq;                 /* queue of sleeping processes          */
uint32  preempt;                /* preemption counter                   */

/*------------------------------------------------------------------------
 * clkinit - initialize the clock and sleep queue at startup
 *------------------------------------------------------------------------
 */
void    clkinit(void)
{
        sleepq = newqueue();    /* allocate a queue to hold the delta   */
                                /* list of sleeping processes           */

        clkticks = 0;           /* start counting one second            */

        /* Add clock interrupt handler to interrupt vector array */

        interruptVector[IRQ_TIMER] = &clkhandler;

        /* Enable clock interrupts */

        enable_irq(IRQ_TIMER);
```

```
        /* Start interval timer */

        clkupdate(CLKCYCS_PER_TICK);
}
```

## 13.15 Interval Timer Management

On the E2100L, the interval timer can only be accessed or controlled through the co-processor. Thus, code to control the timer has been written in assembly language.

Recall that the timer hardware uses the processor clock and interrupts are scheduled on each millisecond. To further understand the timer management code, it is important to know that the hardware uses a count-up approach. The timer always keeps counting and the operating system sets a threshold value *N* units above the current count. When the count reaches the threshold, the timer interrupts.

In theory, emulating a millisecond real-time clock is trivial: when an interrupt occurs, the threshold only needs to be incremented by *N*, where *N* is a constant equal to the number of cycles the timer will accumulate in one millisecond. To see why, recall that the timer hardware always keeps counting. Setting the threshold to *N* beyond the last timer interrupt schedules the next interrupt for exactly one millisecond later.

In practice, however, the simplistic approach of always adding *N* units to the previous threshold may not work. To see why, observe that interrupts might be disabled when the timer expires (e.g., because the processor is handling another device). After the current interrupt finishes, a small amount of additional time passes before the dispatcher calls the clock interrupt handler and the handler updates the timer. Thus, a small amount of time can pass between the timer expiring and the interrupt handler resetting the interval timer. If a millisecond passes, the timer will have already reached a value equal to the previous threshold plus *N*, and the timer will need to wrap around before reaching the threshold. To handle the situation, the timer management code adds *N* to the previous threshold and compares the value to the time count. If the calculated threshold has already passed, the code resets the threshold to the current count plus *N*. File *clkupdate.S* contains the code:

```
/* clkupdate.S - clkupdate, clkcount */

#include <mips.h>

.text
        .align  4
        .globl  clkupdate
        .globl  clkcount

/*------------------------------------------------------------------------
 * clkupdate - update the timer by a specified number of cycles
 *------------------------------------------------------------------------
 */

/* Note: there are two cases
 *  Normal case: COMPARE is increased by N cycles and stored as the
 *               new threshold (N cycles beyond previous threshold)
 *  Abnormal case: the timer has already accumulated more than N cycles
 *               beyond the previous threshold.  Start over by making
 *               the threshold equal to the current count + N
 */

clkupdate:
        mfc0    v0, CP0_COMPARE     /* v0 = COMPARE                      */
        mfc0    v1, CP0_COUNT       /* v1 = COUNT                        */
        addu    v0, v0, a0          /* v0 = COMPARE + cycles             */
        bleu    v0, v1, compare_up  /* v0 <= COUNT, then goto compare_up*/
        mtc0    v0, CP0_COMPARE     /* Update COMPARE                    */
        jr      ra

/* Abnormal case: timer is beyond the next interrupt count; reset       */

compare_up:
        addu    a0, v1, a0          /* a0 = COUNT + cycles               */
        mtc0    a0, CP0_COMPARE     /* COMPARE = a0                      */
        jr      ra

/*------------------------------------------------------------------------
 * clkcount return the count from the free-running clock
 *------------------------------------------------------------------------
 */
clkcount:
        mfc0 v0, CP0_COUNT
        jr   ra
```

## 13.16 Perspective

Clock and timer management are both technically and intellectually challenging. On the one hand, because clock or timer interrupts occur frequently and have high priority, the total time a CPU spends executing clock interrupts is large and other interrupts are prevented. Thus, the code for an interrupt handler must be optimized to minimize the time taken to handle a given interrupt. On the other hand, an operating system that allows processes to request timed events can schedule many events to occur at exactly the same time, which means that the time taken for a given interrupt can be arbitrarily long. The conflict can become especially important in real-time embedded systems where the processor is relatively slow and other devices need prompt service.

Most systems allow arbitrary events to be scheduled and defer processing when multiple events collide — a cell phone may not update the display exactly when an application starts, or a text message may take longer to deliver when multiple applications are running. The intellectual questions are: how can an operating system best provide the illusion of precise timing within hardware constraints and inform users when requests cannot be serviced? Should events be assigned priorities? If so, how should event priorities interact with scheduling priorities? There are no easy answers.

## 13.17 Summary

A real-time clock interrupts the CPU at regular intervals. The design described uses a timer to emulate real-time clock interrupts. The interrupt handler processes the interrupt and resets the timer for the next interrupt.

The operating system uses the clock to handle preemption and process delay. A preemption event, scheduled every time the system switches context, forces a call to the scheduler after a process has used the processor for *QUANTUM* clock ticks. Preemption guarantees that no process uses the CPU forever, and enforces the scheduling policy by insuring round-robin service among equal-priority processes.

A wakeup event, scheduled when a process requests a timed delay, causes the running process to enqueue itself on the delta list of sleeping processes. The interrupt handler awakens a sleeping process when its delay expires by moving the process back to the ready list and rescheduling. A delta list provides a very efficient way to manage sleeping processes.

## EXERCISES

**13.1** Modify the code to generate clock interrupts ten times faster, and arrange for the clock interrupt handler to ignore nine interrupts before processing one. How much extra overhead do the additional interrupts generate?

**13.2** Conduct an experiment to determine whether the system ever misses a clock interrupt and, if so, how often interrupts are missed. When an interrupt occurs, read the accumulated count from the timer and compare to the "comparison" value to see how many additional cycles have accumulated beyond those that were expected.

**13.3** Trace the series of calls starting with a clock interrupt that awakens two sleeping processes, one of which has higher priority than the currently executing process.

**13.4** Explain what can fail if *QUANTUM* is set to 1. Hint: consider switching back to a process that was suspended by *resched* while processing an interrupt.

**13.5** Does *sleepms*(3) guarantee a minimum delay of 3 milliseconds, an exact delay of 3 milliseconds, or a maximum delay of 3 milliseconds?

**13.6** Carefully consider the code for *kill* and identify a problem that is caused when *kill* removes a process from the queue of sleeping processes. Rewrite *kill* to correct the problem.

**13.7** What might happen if *wakeup* calls *wait*?

**13.8** An operating system that attempts to record the exact amount of processor time a process consumes faces the following problem: when an interrupt occurs, it is most convenient to let the current process execute the interrupt routine even though the interrupt is unlikely to be related to the current process. Investigate how operating systems charge the cost of executing interrupt routines like *wakeup* to the processes that are affected.

**13.9** If the code in this chapter is ported to an identical processor that runs at a higher clock rate, what will happen? Why?

**13.10** Design an experiment to see if preemption ever causes the system to reschedule. Be careful: the presence of a separate process testing a variable or performing I/O can interfere with the experiment by generating calls to *resched*.

**13.11** Suppose a system contains three processes: a low-priority process, $L$, that is sleeping, and two high-priority processes, $H_1$ and $H_2$, that are eligible to execute. Further suppose that immediately after the scheduler switches to process $H_1$, a clock interrupt occurs, process $L$ becomes ready, and the interrupt handler calls *resched*. Although $L$ will not run, *resched* will switch from $H_1$ to $H_2$ without giving $H_1$ its quantum. Propose a modification to *resched* that insures a process will not lose control of the processor unless a higher-priority process becomes ready or its time slice expires.

# NOTES

# Chapter Contents

# 14

# Device–Independent Input and Output

*We have been left so much to our own devices — after a while, one welcomes the uncertainty of being left to other people's.*

— Tom Stoppard

## 14.1 Introduction

Earlier chapters explain concurrent process support and memory management. Chapter 12 discusses the key concept of interrupts. The chapter describes interrupt processing, gives an architecture for interrupt code, and explains the relationship between interrupt handling and concurrent processes. Chapter 13 expands the discussion of interrupts by showing how real-time clock interrupts can be used to implement preemption and process delay.

This chapter takes a broader look at how an operating system implements I/O. The chapter explains the conceptual basis for building I/O abstractions, and presents an architecture for a general-purpose I/O facility. The chapter shows how processes can transfer data to or from a device without understanding the underlying hardware. It defines a general model, and explains how the model incorporates device-independent I/O functions. Finally, the chapter examines an efficient implementation of an I/O subsystem.

## 14.2 Conceptual Organization Of I/O And Device Drivers

Operating systems control and manage input and output (I/O) devices for three reasons. First, because most device hardware uses a low-level interface, the software interface is complex. Second, because a device is a shared resource, an operating system provides access according to policies that make sharing fair and safe. Third, an operating system defines a high-level interface that hides details and allows a programmer to use a consistent and uniform set of operations when interacting with devices.

The I/O subsystem can be divided into three conceptual pieces: an abstract interface consisting of high-level I/O functions that processes use to perform I/O, a set of physical devices, and *device driver* software that connects the two. Figure 14.1 illustrates the organization.

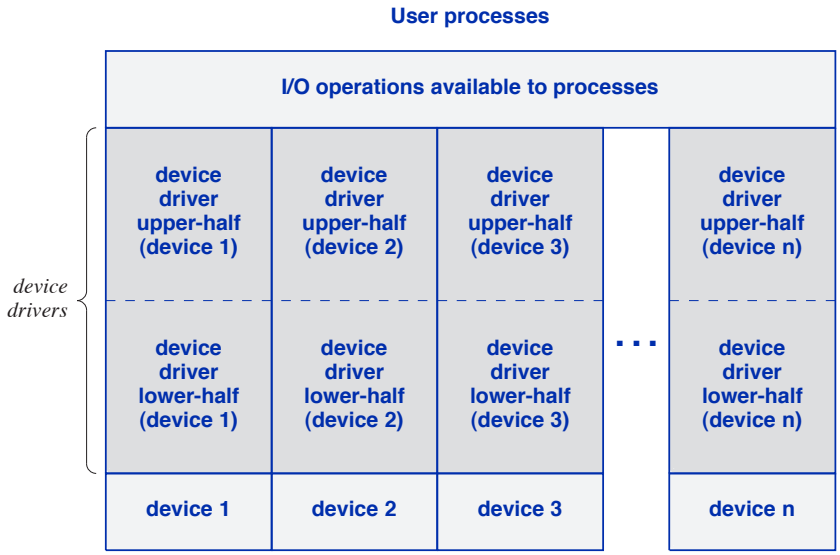



**Figure 14.1** The conceptual organization of the I/O subsystem with device driver software between processes and the underlying device hardware.

As the figure indicates, device driver software bridges the gap between high-level, concurrent processes and low-level hardware. We will see that each driver is divided into two conceptual pieces: an *upper half* and a *lower half*. Functions in the upper half are invoked when a process requests I/O. Upper-half functions implement operations such as *read* and *write* by transferring data to or from a process. The lower half contains functions that are invoked by interrupts. When a device interrupts, the interrupt dispatcher invokes a lower-half handler function. The handler services the interrupt, interacts with the device to transfer data, and may start an additional I/O operation.

## 14.3 Interface And Driver Abstractions

The ultimate goal of an operating system designer lies in creating convenient programming abstractions and finding efficient implementations. With respect to I/O, there are two aspects:

- Interface abstraction
- Driver abstractions

*Interface abstraction*. The question arises: what I/O interface should an operating system supply to processes? There are several possibilities, and the choice represents a tradeoff among goals of flexibility, simplicity, efficiency, and generality. To understand the scope of the problem, consider Figure 14.2, which lists a set of example devices and the types of operations appropriate for each.

| Device | I/O paradigm |
|---|---|
| hard drive | move to a position and transfer a block of data |
| keyboard | accept individual characters as entered |
| printer | transfer an entire document to be printed |
| audio output | transfer a continuous stream of encoded audio |
| wireless network | send or receive a single network packet |

**Figure 14.2** Example devices and the paradigm that each device uses.

Early operating systems provided a set of I/O operations for each individual hardware device. Unfortunately, building device-specific information into software means the software must be changed when an I/O device is replaced by an equivalent device from another vendor. A slightly more general approach defines a set of operations for each type of device, and requires the operating system to perform the appropriate low-level operations on a given device. For example, an operating system can offer abstract functions *send_network_packet* and *receive_network_packet* that can be used to transfer network packets over any type of network. A third approach originated in Multics and was popularized by Unix: choose a small set of abstract I/O operations that are sufficient for all I/O.

*Driver abstractions*. We think of the second category of abstraction as providing semantics. One of the most important semantic design questions focuses on synchrony: does a process block while waiting for an I/O operation to complete? A *synchronous* interface, similar to the one described earlier, provides blocking operations. For example, to request data from a keyboard in a synchronous system, a process invokes an

upper-half routine that blocks the process until a user presses a key. Once a user makes a keystroke, the device interrupts and the dispatcher invokes a lower-half routine that acts as a handler. The handler unblocks a waiting process and reschedules to allow the process to run. In contrast, an *asynchronous* I/O interface allows a process to continue executing after the process initiates an I/O operation. When the I/O completes, the driver must inform the requesting process (e.g., by invoking the *event handler* function associated with the process). We can summarize:

> *When using a synchronous I/O interface, a process is blocked until the operation completes. When using an asynchronous I/O interface, a process continues to execute and is notified when the operation completes.*

Each approach has advantages. An asynchronous interface is useful in situations where a programmer needs to control the overlap of I/O and computation. A synchronous approach has the advantage of being easier to program.

Another design issue arises from the format of data and the size of transfers. Two questions arise. First, will data be transferred in blocks or bytes? Second, how much data can be transferred in a single operation? Observe that some devices transfer individual data bytes, some transfer a variable-size chunk of data (such as a network packet or a line of text), and others operate on fixed-size blocks of data. Because a general-purpose operating system must handle a variety of I/O devices, an I/O interface may require both single-byte transfers as well as multi-byte transfers.

A final design question arises from the parameters that a driver supplies and the way a driver interprets individual operations. For example, does a process specify a location on disk and then repeatedly request the next disk block, or does the process specify a block number in each request? The example device driver illustrates how parameters can be used.

The key idea is:

> *In a modern operating system, the I/O interface and device drivers are designed to hide device details and present a programmer with convenient, high-level abstractions.*

## 14.4 An Example I/O Interface

Experience has shown that a small set of I/O functions is both sufficient and convenient. Thus, our example system contains an I/O subsystem with nine abstract I/O operations that are used for all input and output. The operations have been derived from the I/O facilities in the Unix operating system. Figure 14.3 lists the operations and the purpose of each.

| Operation | Purpose |
|---|---|
| close | Terminate use of a device |
| control | Perform operations other than data transfer |
| getc | Input a single byte of data |
| init | Initialize the device at system startup |
| open | Prepare the device for use |
| putc | Output a single byte of data |
| read | Input multiple bytes of data |
| seek | Move to specific data (usually a disk) |
| write | Output multiple bytes of data |

**Figure 14.3** The set of abstract I/O interface operations used in Xinu.

## 14.5 The Open-Read-Write-Close Paradigm

Like the programming interface in many operating systems, the example I/O interface follows an *open-read-write-close* paradigm. That is, before it can perform I/O, a process must *open* a specific device. Once it has been opened, a device is ready for the process to call *read* to obtain input or *write* to send output. Finally, once it has finished using a device, the process calls *close* to terminate use.

To summarize:

> *The open-read-write-close paradigm requires a process to open a device before use and close a device after use.*

*Open* and *close* allow the operating system to manage devices that require exclusive use, prepare a device for data transfer, and stop a device after transfer has ended. Closing a device may be useful, for example, if a device needs to be powered down or placed in a standby state when not in use. *Read* and *write* handle the transfer of multiple data bytes to or from a buffer in memory. *Getc* and *putc* form a counterpart for the transfer of a single byte (usually a character). *Control* allows a program to control a device or a device driver (e.g., to check supplies in a printer or select the channel on a wireless radio). *Seek* is a special case of *control* that applies to randomly accessible storage devices, such as disks. Finally, *init* initializes the device and driver at system startup.

Consider how the operations apply to a console window. *Getc* reads the next character from the keyboard, and *putc* displays one character in the console window. *Write* can display multiple characters with one call, and *read* can read a specified number of characters (or all that have been entered, depending on its arguments). Finally, *control* allows the program to change parameters in the driver to control such things as whether the system stops echoing characters as a password is entered.

## 14.6 Bindings For I/O Operations And Device Names

How can an abstract operation such as *read* act on an underlying hardware device? The answer lies in a binding. When a process invokes a high-level operation, the operating system maps the call to a device driver function. For example, if a process calls *read* on a console device, the operating system passes the call to a function in the console device driver that implements *read*. In doing so, the operating system hides both the hardware and device driver details from application processes and presents an abstract version of devices. By using a single abstract device for a keyboard and a window on the display, an operating system can hide the fact that the underlying hardware consists of two separate devices. Furthermore, an operating system can hide device details by presenting the same high-level abstraction for the hardware from multiple vendors. The point is:

> *An operating system creates a virtual I/O environment — a process can only perceive peripheral devices through the abstractions that the interface and device drivers provide.*

In addition to mapping abstract I/O operations onto driver routines, an operating system must map device names onto devices. A variety of mappings have been used. Early systems required a programmer to embed device names in source code. Later systems arranged for programs to use small integers to identify devices, and allowed a command interpreter to link each integer with a specific device when the application was launched. Many modern systems embed devices in a file naming hierarchy, allowing an application to use a symbolic name for each device.

Early and late binding each have advantages. An operating system that waits until run-time to bind the name of an abstract device to a real device and a set of abstract operations to device driver functions is flexible. However, such late binding systems incur more computational overhead, making them impractical in the smallest embedded systems. At the other extreme, early binding requires device information to be specified when an application is written. Thus, the essence of I/O design consists of synthesizing a binding mechanism that allows maximum flexibility within the required performance bounds.

Our example system uses an approach that is typical of small embedded systems: information about devices is specified before the operating system is compiled. For each device, the operating system knows exactly which driver functions correspond to

each of the abstract I/O operations. In addition, the operating system knows the underlying hardware device to which each abstract device corresponds. Thus, the operating system must be recompiled whenever a new device is added or when an existing device is removed. Because application code does not contain information about specific device hardware, application code can be ported from one system to another easily. For example, an application that only performs I/O operations on a *CONSOLE* serial port will work on any Xinu system that offers a *CONSOLE* device and the appropriate driver, independent of the physical device hardware and interrupt structure.

## 14.7 Device Names In Xinu

In Xinu, the system designer must specify a set of abstract devices when the system is configured. The configuration program assigns each device name a unique integer value known as a *device descriptor*. For example, if a designer specifies a device name *CONSOLE*, the configuration program might assign descriptor zero. The configuration program produces a header file that contains *#define* statements for each name. Thus, once the header file has been included, a programmer can reference *CONSOLE* in the code. For example, if *CONSOLE* has been assigned descriptor zero, the call:

```
read(CONSOLE, buf, 100);
```

is equivalent to:

```
read(0, buf, 100);
```

To summarize:

> *Xinu uses a static binding for device names. Each device name is bound to an integer descriptor at configuration time before the operating system is compiled.*

## 14.8 The Concept Of A Device Switch Table

Each time a process invokes a high-level I/O operation such as *read* or *write*, the operating system must forward the call to the appropriate driver function. To make the implementation efficient, Xinu uses an array known as a *device switch table*. The integer descriptor assigned to a device is an index into the device switch table. To understand the arrangement, imagine a two-dimensional array. Conceptually, each row of the array corresponds to a device, and each column corresponds to an abstract operation. An entry in the array specifies the driver function to use to perform the operation.

For example, suppose a system contains three devices defined as follows:

- CONSOLE, a serial device used to send and receive characters
- ETHER, an Ethernet interface device
- DISK, a hard drive

Figure 14.4 illustrates part of a device switch table that has a row for each of the three devices and a column for each I/O operation. Items in the table represent the names of driver functions that perform the operation given by the column on the device given by the row.

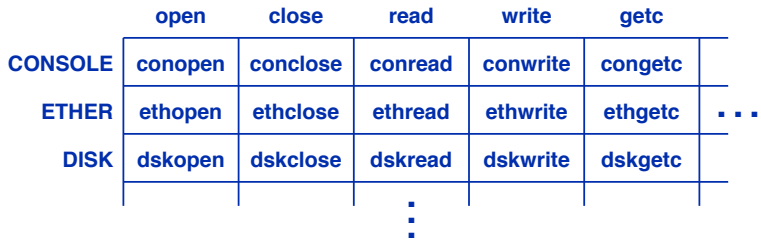



**Figure 14.4** Conceptual organization of the device switch table with one row per device and one column per abstract I/O operation.

As an example, suppose a process invokes the *write* operation on the *CONSOLE* device. The operating system goes to the row of the table that corresponds to the *CONSOLE* device, finds the column that corresponds to the *write* operation, and calls the function named in the entry: *conwrite*.

In essence, each row of the device switch table defines how the I/O operations apply to a single device, which means I/O semantics can change dramatically among devices. For example, when it is applied to a *DISK* device, a *read* might transfer a block of 512 bytes of data. However, when it is applied to a *CONSOLE* device, *read* might transfer a line of characters that the user has entered.

The most significant aspect of the device switch table arises from the ability to define a uniform abstraction across multiple physical devices. For example, suppose a computer contains a disk that uses 1 Kbyte sectors and a disk that uses 4 Kbyte sectors. Drivers for the two disks can present an identical interface to applications, while hiding the differences in the underlying hardware. That is, a driver can always transfer 4 Kbytes to a user, and convert each transfer into four 1 Kbyte disk transfers.

## 14.9 Multiple Copies Of A Device And Shared Drivers

Suppose a given computer has two devices that use identical hardware. Does the operating system need two separate copies of the device driver? No. The system contains one copy of each driver routine and uses a parameter to distinguish between the two devices. Parameters are kept in columns of the device switch table in addition to the functions that Figure 14.4 illustrates. For example, if a system contains two Ethernet interfaces, each will have its own row in the device switch table. Most entries in the two rows will be identical. However, one column will specify a unique *Control and Status Register* (*CSR*) address for each device. When it invokes a driver function, the

system passes an argument that contains a pointer to the row in the device switch table for the device. Thus, a driver function can apply the operation to the correct device. The point is:

> *Instead of creating a device driver for each physical device, an operating system maintains a single copy of the driver for each type of device and supplies an argument that permits the driver to distinguish among multiple copies of the physical hardware.*

A look at the definition of the device switch table, *devtab*, will clarify the details. Structure *dentry* defines the format of entries in the table; the declaration can be found in file *conf.h*.†

```
/* conf.h (GENERATED FILE; DO NOT EDIT) */

/* Device switch table declarations */

/* Device table entry */
struct  dentry  {
        int32   dvnum;
        int32   dvminor;
        char    *dvname;
        devcall (*dvinit) (struct dentry *);
        devcall (*dvopen) (struct dentry *, char *, char *);
        devcall (*dvclose)(struct dentry *);
        devcall (*dvread) (struct dentry *, void *, uint32);
        devcall (*dvwrite)(struct dentry *, void *, uint32);
        devcall (*dvseek) (struct dentry *, int32);
        devcall (*dvgetc) (struct dentry *);
        devcall (*dvputc) (struct dentry *, char);
        devcall (*dvcntl) (struct dentry *, int32, int32, int32);
        void    *dvcsr;
        void    (*dvintr)(void);
        byte    dvirq;
};

extern  struct  dentry  devtab[]; /* one entry per device */

/* Device name definitions */

#define CONSOLE     0       /* type tty      */
#define NOTADEV     1       /* type null     */
#define ETHER0      2       /* type eth      */
```

†File *conf.h* also contains *#define* statements that define constants used throughout the system; Chapter 24 describes Xinu configuration and explains how the constants appear.

```
#define RFILESYS    3       /* type rfs      */
#define RFILE0      4       /* type rfl      */
#define RFILE1      5       /* type rfl      */
#define RFILE2      6       /* type rfl      */
#define RFILE3      7       /* type rfl      */
#define RFILE4      8       /* type rfl      */
#define RFILE5      9       /* type rfl      */
#define RDISK       10       /* type rds      */
#define LFILESYS    11       /* type lfs      */
#define LFILE0      12       /* type lfl      */
#define LFILE1      13       /* type lfl      */
#define LFILE2      14       /* type lfl      */
#define LFILE3      15       /* type lfl      */
#define LFILE4      16       /* type lfl      */
#define LFILE5      17       /* type lfl      */
#define TESTDISK    18       /* type ram      */
#define NAMESPACE   19       /* type nam      */

/* Control block sizes */

#define Nnull   1
#define Ntty    1
#define Neth    1
#define Nrfs    1
#define Nrfl    6
#define Nrds    1
#define Nram    1
#define Nlfs    1
#define Nlfl    6
#define Nnam    1

#define DEVMAXNAME 24
#define NDEVS 20


/* Configuration and Size Constants */

#define NPROC        100        /* number of user processes             */
#define NSEM         100        /* number of semaphores                 */
#define IRQ_TIMER    IRQ_HW5    /* timer IRQ is wired to hardware 5     */
#define IRQ_ATH_MISC IRQ_HW4    /* Misc. IRQ is wired to hardware 4     */
#define MAXADDR      0x02000000 /* 32 MB of RAM                         */
#define CLKFREQ      200000000  /* 200 MHz clock                        */
#define FLASH_BASE   0xBD000000 /* Flash ROM device                     */

#define LF_DISK_DEV     TESTDISK
```

Each entry in *devtab* corresponds to a single device. The entry specifies the address of functions that constitute the driver for the device, the device CSR address, and other information used by the driver. Fields *dvinit*, *dvopen*, *dvclose*, *dvread*, *dvwrite*, *dvseek*, *dvgetc*, *dvputc*, and *dvcntl* hold the addresses of driver routines corresponding to the high-level operations. Field *dvminor* contains an integer index into the control block array for the device. A minor device number is essential if the underlying hardware includes a set of identical hardware devices — the minor number means a driver can have a separate control block entry for each device. Field *dvcsr* contains the hardware CSR address for the device. Each entry in the control block array holds additional information for the particular instance of the device and the driver. The contents of a control block depend on the device, but may include such things as input or output buffers, device status information (e.g., whether a wireless networking device is currently in contact with another wireless device), or accounting information (e.g., the total amount of data received over a network device since the system booted).

## 14.10 The Implementation Of High-Level I/O Operations

Because it isolates high-level I/O operations from underlying details, the device switch table allows high-level functions to be created before any device drivers have been written. One of the chief benefits of such a strategy arises because a programmer can build pieces of the I/O system without requiring specific hardware devices to be present.

The example system contains a function for each high-level I/O operation. Thus, the system contains functions *open*, *close*, *read*, *write*, *getc*, *putc*, and so on. However, a function such as *read* does not perform I/O. Instead, each high-level I/O function operates *indirectly*: the function uses the device switch table to find and invoke the appropriate low-level device driver routine to perform the requested function. The point is:

> *Instead of performing I/O, high-level functions such as* read *and* write *use a level of indirection to invoke a low-level driver function for the specified device.*

An examination of the code will clarify the concept. Consider the *read* function found in file *read.c*:

```
/* read.c - read */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  read  -  read one or more bytes from a device
 *------------------------------------------------------------------------
 */
syscall read(
          did32         descrp,         /* descriptor for device        */
          char          *buffer,        /* address of buffer            */
          uint32        count           /* length of buffer             */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvread) (devptr, buffer, count);
        restore(mask);
        return retval;
}
```

The arguments to *read* consist of a *device descriptor*, the address of a buffer, and an integer that gives the maximum number of bytes to read. *Read* uses the device descriptor, *descrp*, as an index into *devtab*, and assigns pointer *devptr* the address of the device switch table entry. The *return* statement contains code that performs the task of invoking the underlying device driver function and returning the result to the function that called *read*. The code:

```
(*devptr->dvread) (devptr, buffer, count)
```

performs the indirect function call. That is, the code invokes the driver function given by field *dvread* in the device switch table entry, passing the function three arguments: the address of the *devtab* entry, *devptr*, the buffer address, *buffer*, and a count of characters to read, *count*.

## 14.11 Other High-Level I/O Functions

The remaining high-level transfer and control functions operate exactly as *read*: they use the device switch table to select the appropriate low-level driver function, invoke the function, and return the result to the caller. Code for each function is shown below.

```
/* control.c - control */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  control  -  control a device or a driver (e.g., set the driver mode)
 *------------------------------------------------------------------------
 */
syscall control(
          did32         descrp,         /* descriptor for device        */
          int32         func,           /* specific control function    */
          int32         arg1,           /* specific argument for func   */
          int32         arg2            /* specific argument for func   */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvcntl) (devptr, func, arg1, arg2);
        restore(mask);
        return retval;
}
```

```
/* getc.c - getc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getc  -  obtain one byte from a device
 *------------------------------------------------------------------------
 */
syscall getc(
          did32         descrp          /* descriptor for device        */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvgetc) (devptr);
        restore(mask);
        return retval;
}


/* putc.c - putc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  putc  -  send one character of data (byte) to a device
 *------------------------------------------------------------------------
 */
syscall putc(
          did32         descrp,         /* descriptor for device        */
          char          ch              /* character to send            */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
```

```
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvputc) (devptr, ch);
        restore(mask);
        return retval;
}


/* seek.c - seek */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  seek  -  position a random access device
 *------------------------------------------------------------------------
 */
syscall seek(
          did32         descrp,         /* descriptor for device        */
          uint32        pos             /* position                     */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvseek) (devptr, pos);
        restore(mask);
        return retval;
}
```

```
/* write.c - write */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  write  -  write one or more bytes to a device
 *------------------------------------------------------------------------
 */
syscall write(
          did32         descrp,         /* descriptor for device        */
          char          *buffer,        /* address of buffer            */
          uint32        count           /* length of buffer             */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvwrite) (devptr, buffer, count);
        restore(mask);
        return retval;
}
```

The functions listed above are designed to allow user processes to access I/O devices. In addition, the system provides one high-level I/O function that is intended only for the operating system to use: *init*. We will see that when it boots, the operating system calls *init* for each device. Like the other I/O functions, *init* uses the device switch table to invoke the appropriate low-level driver function. Thus, the initialization function in each driver can initialize the hardware device, if necessary, and can also initialize the data structures used by the driver (e.g., buffers and semaphores). We will see examples of driver initialization later. For now, it is sufficient to understand that *init* follows the same approach as other I/O routines:

```
/* init.c - init */

#include <xinu.h>

/*------------------------------------------------------------------------
```

```
 *  init  -  initialize a device and its driver
 *------------------------------------------------------------------------
 */
syscall init(
          did32         descrp          /* descriptor for device        */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvinit) (devptr);
        restore(mask);
        return retval;
}
```

## 14.12 Open, Close, And Reference Counting

Functions *open* and *close* operate similar to other I/O functions by using the device switch table to call the appropriate driver function. One motivation for using *open* and *close* arises from their ability to establish ownership of a device or prepare a device for use. For example, if a device requires exclusive access, *open* can block a subsequent user until the device becomes free. As another example, consider a system that saves power by keeping a disk device idle when the device is not in use. Although a designer could arrange to use the *control* function to start or stop a disk, *open* and *close* are more convenient. Thus, a disk can be powered on when a process calls *open*, and powered off when a process calls *close*.

Although a small embedded system might choose to power down a disk whenever a process calls *close* on the device, larger systems need a more sophisticated mechanism because multiple processes can use a device simultaneously. Thus, most drivers employ a technique known as *reference counting*. That is, a driver maintains an integer variable that counts the number of processes using the device. During initialization, the reference count is set to zero. Whenever a process calls *open*, the driver increments the reference count, and whenever a process calls *close*, the driver decrements the reference count. When the reference count reaches zero, the driver powers down the device.

The code for *open* and *close* follows the same approach as the code for other high-level I/O functions:

```
/* open.c - open */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  open  -  open a device (some devices ignore name and mode parameters)
 *------------------------------------------------------------------------
 */
syscall open(
          did32         descrp,         /* descriptor for device        */
          char          *name,          /* name to use, if any          */
          char          *mode           /* mode for device, if any      */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct dentry   *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvopen) (devptr, name, mode);
        restore(mask);
        return retval;
}


/* close.c - close */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  close  -  close a device
 *------------------------------------------------------------------------
 */
syscall close(
          did32         descrp          /* descriptor for device        */
        )
{
        intmask         mask;           /* saved interrupt mask         */
        struct  dentry  *devptr;        /* entry in device switch table */
        int32           retval;         /* value to return to caller    */
```

```
        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvclose) (devptr);
        restore(mask);
        return retval;
}
```

## 14.13 Null And Error Entries In Devtab

An interesting dilemma arises from the way I/O functions operate. On one hand, high-level functions, such as *read* and *write*, use entries in *devtab* without checking whether the entries are valid. Thus, a function must be supplied for every I/O operation for each device. On the other hand, an operation may not be meaningful on all devices. For example, *seek* is not an operation that can be performed on a serial device, and *getc* is not meaningful on a network device that delivers packets. Furthermore, a designer may choose to ignore an operation on a particular device (e.g., a designer may choose to leave the CONSOLE device open at all times, which means the *close* operation has no effect).

What value can be used in *devtab* for operations that are not meaningful? The answer lies in two special routines that can be used to fill in entries of *devtab* that have no driver functions:

- ionull — return OK without performing any action
- ioerr — return SYSERR without performing any action

By convention, entries filled with *ioerr* should never be called; they signify an illegal operation. Entries for unnecessary, but otherwise innocuous operations (e.g., *open* for a terminal device), point to function *ionull*. The code for each of the two functions is trivial.

```
/* ionull.c - ionull */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ionull  -  do nothing (used for "don't care" entries in devtab)
 *------------------------------------------------------------------------
 */
devcall ionull(void)
{
        return OK;
}


/* ioerr.c - ioerr */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ioerr  -  return an error status (used for "error" entries in devtab)
 *------------------------------------------------------------------------
 */
devcall ioerr(void)
{
        return SYSERR;
}
```

## 14.14 Initialization Of The I/O System

How is a device switch table initialized? How are driver functions installed? In a large, complex operating system, device drivers can be managed dynamically. Thus, when a user plugs in a new device, the operating system can identify the hardware and search for an appropriate driver, and install the driver without rebooting.

A small embedded system does not have a collection of drivers available on secondary storage, and may not have sufficient computational resources to install drivers at run time. Thus, most embedded systems use a static device configuration in which the set of devices and the set of device drivers are specified when the system is compiled. Our example follows the static approach by requiring the system designer to specify a set of devices and the set of low-level driver functions that constitute each driver. Instead of forcing a programmer to enter explicit declarations for the entire device switch table, however, a separate application program is used that reads a configuration file and generates a C file that contains a declaration of *devtab* with an initial value for each field. The point is:

*Small embedded systems use a static device specification in which a designer specifies the set of devices plus the device driver functions for each; a configuration program can generate code that assigns a value to each field of the device switch table.*

File *conf.c* contains an example of the C code generated by the configuration program. For now, it is sufficient to examine one of the entries in *devtab* and observe how each field is initialized.

```
/* conf.c (GENERATED FILE; DO NOT EDIT) */

#include <xinu.h>


extern  devcall ioerr(void);
extern  devcall ionull(void);

/* Device independent I/O switch */

struct  dentry  devtab[NDEVS] =
{
/**
 * Format of entries is:
 * dev-number, minor-number, dev-name,
 * init, open, close,
 * read, write, seek,
 * getc, putc, control,
 * dev-csr-address, intr-handler, irq
 */

/* CONSOLE is tty */
        { 0, 0, "CONSOLE",
          (void *)ttyInit, (void *)ionull, (void *)ionull,
          (void *)ttyRead, (void *)ttyWrite, (void *)ioerr,
          (void *)ttyGetc, (void *)ttyPutc, (void *)ttyControl,
          (void *)0xb8020000, (void *)ttyInterrupt, 11 },

/* NOTADEV is null */
        { 1, 0, "NOTADEV",
          (void *)ionull, (void *)ionull, (void *)ionull,
          (void *)ionull, (void *)ionull, (void *)ioerr,
          (void *)ionull, (void *)ionull, (void *)ioerr,
          (void *)0x0, (void *)ioerr, 0 },
```

```
/* ETHER0 is eth */
        { 2, 0, "ETHER0",
          (void *)ethInit, (void *)ethOpen, (void *)ioerr,
          (void *)ethRead, (void *)ethWrite, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ethControl,
          (void *)0xb9000000, (void *)ethInterrupt, 4 },

/* RFILESYS is rfs */
        { 3, 0, "RFILESYS",
          (void *)rfsInit, (void *)rfsOpen, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)rfsControl,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE0 is rfl */
        { 4, 0, "RFILE0",
          (void *)rflInit, (void *)ioerr, (void *)rflClose,
          (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
          (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE1 is rfl */
        { 5, 1, "RFILE1",
          (void *)rflInit, (void *)ioerr, (void *)rflClose,
          (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
          (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE2 is rfl */
        { 6, 2, "RFILE2",
          (void *)rflInit, (void *)ioerr, (void *)rflClose,
          (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
          (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE3 is rfl */
        { 7, 3, "RFILE3",
          (void *)rflInit, (void *)ioerr, (void *)rflClose,
          (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
          (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE4 is rfl */
        { 8, 4, "RFILE4",
          (void *)rflInit, (void *)ioerr, (void *)rflClose,
```

```
          (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
          (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE5 is rfl */
        { 9, 5, "RFILE5",
          (void *)rflInit, (void *)ioerr, (void *)rflClose,
          (void *)rflRead, (void *)rflWrite, (void *)rflSeek,
          (void *)rflGetc, (void *)rflPutc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RDISK is rds */
        { 10, 0, "RDISK",
          (void *)rdsInit, (void *)rdsOpen, (void *)rdsClose,
          (void *)rdsRead, (void *)rdsWrite, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)rdsControl,
          (void *)0x0, (void *)ionull, 0 },

/* LFILESYS is lfs */
        { 11, 0, "LFILESYS",
          (void *)lfsInit, (void *)lfsOpen, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE0 is lfl */
        { 12, 0, "LFILE0",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE1 is lfl */
        { 13, 1, "LFILE1",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE2 is lfl */
        { 14, 2, "LFILE2",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },
```

```
/* LFILE3 is lfl */
        { 15, 3, "LFILE3",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE4 is lfl */
        { 16, 4, "LFILE4",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE5 is lfl */
        { 17, 5, "LFILE5",
          (void *)lflInit, (void *)ioerr, (void *)lflClose,
          (void *)lflRead, (void *)lflWrite, (void *)lflSeek,
          (void *)lflGetc, (void *)lflPutc, (void *)lflControl,
          (void *)0x0, (void *)ionull, 0 },

/* TESTDISK is ram */
        { 18, 0, "TESTDISK",
          (void *)ramInit, (void *)ramOpen, (void *)ramClose,
          (void *)ramRead, (void *)ramWrite, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* NAMESPACE is nam */
        { 19, 0, "NAMESPACE",
          (void *)namInit, (void *)namOpen, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)0x0, (void *)ioerr, 0 }
};
```

## 14.15 Perspective

Device-independent I/O is now an integral part of main-stream computing, and the advantages seem obvious. However, it took decades for the computing community to reach consensus on device-independent I/O and to devise a set of primitives. Some of the contention arose because programming languages each define a set of I/O abstractions. For example, FORTRAN used device numbers and required a mechanism that

could bind each number to an I/O device or file. Operating system designers wanted to accommodate all languages because a large volume of code has been written in each. So, the question arises: have we chosen the best set of device-independent I/O functions or have we merely become so accustomed to using them that we fail to look for alternatives?

## 14.16 Summary

An operating system hides the details of peripheral devices, and provides a set of abstract, device-independent functions that can be used to perform I/O. The example system uses nine abstract functions: *open*, *close*, *control*, *getc*, *putc*, *read*, *write*, *seek*, and an initialization function, *init*. In our design, each of the I/O primitives operates *synchronously*, delaying a calling process until the request has been satisfied (e.g., function *read* delays the calling process until data has arrived).

The example system defines an abstract device name (such as *CONSOLE*) for each device, and assigns the device a unique integer device descriptor. The system uses a device switch table to bind a descriptor to a specific device at run-time. Conceptually, the device switch table contains one row for each device and one column for each abstract I/O operation; additional columns point to a control block for the device, and a minor device number is used to distinguish among multiple copies of a physical device. A high-level I/O operation, such as *read* or *write*, uses the device switch table to invoke the device driver function that performs the requested operation on the specified device. Individual drivers interpret the calls in a way meaningful to a particular device; if an operation makes no sense when applied to a particular device, the device switch table is configured to invoke function *ioerr*, which returns an error code.

## EXERCISES

**14.1** Identify the set of abstract I/O operations available in Linux.

**14.2** Find a system that uses *asynchronous* I/O, and identify the mechanism by which a running program is notified when the operation completes. Which approach, synchronous or asynchronous, makes it easier to program? Explain.

**14.3** The chapter discusses two separate bindings: the binding from a device name (e.g., *CONSOLE*) to a descriptor (e.g., 0) and the binding from a device descriptor to a specific hardware device. Explain how Linux performs the two bindings.

**14.4** Consider the implementation of device names in the example code. Is it possible to write a program that allows a user to enter a device name (e.g., *CONSOLE*), and then open the device? Why or why not?

**14.5** Assume that in the course of debugging you begin to suspect that a process is making incorrect calls to high-level I/O functions (e.g., calling *seek* on a device for which the operation makes no sense). How can you make a quick change to the code to intercept such errors and display the process ID of the offending process? (Make the change without recompiling the source code.)

**14.6** Are the abstract I/O operations presented in the chapter sufficient for all I/O operations? Explain. (Hint: consider socket functions found in Unix.)

**14.7** Xinu defines the device subsystem as the fundmental I/O abstraction and merges files into the device system. Unix systems define the file system as the fundamental abstraction and merge devices into the file system. Compare the two approaches and list the advantages of each.

# NOTES

# Chapter Contents

# 15

# An Example Device Driver

*It's hard to find a good driver these days, one with character and style.*

— Unknown

## 15.1 Introduction

Chapters in this section of the text explore the general structure of an I/O system, including interrupt processing and real-time clock management. The previous chapter presents the organization of the I/O subsystem, a set of abstract I/O operations, and an efficient implementation using a device switch table.

This chapter continues the exploration of I/O. The chapter explains how a driver can define an I/O service at a high level of abstraction that is independent of the underlying hardware. The chapter also elaborates on the conceptual division of a device driver into upper and lower halves by explaining how the two halves share data structures, such as buffers, and how they communicate. Finally, the chapter shows the details of a particular example: a driver for an asynchronous character-oriented serial device.

## 15.2 The Tty Abstraction

Xinu uses the name *tty* to refer to the abstraction of an interface used with character-oriented serial devices such as a serial interface or a keyboard and text window.† In broad terms, a tty device supports two-way communication: a process can send characters to the output side and/or receive characters from the input side. Although the underlying serial hardware mechanism operates the input and output in-

†The name *tty* is taken from early Unix systems that used an ASCII Teletype device that consisted of a keyboard and an associated printer mechanism.

dependently, the tty abstraction allows the two to be connected. For example, our tty driver supports *character echo*, which means that the input side of the driver can be configured to transmit a copy of each incoming character to the output. Echo is especially important when a user is typing on a keyboard and expects to see characters displayed on a screen as keys are pressed.

The tty abstraction illustrates an important feature of many device drivers: multiple *modes* that can be selected at run-time. In our tty driver, the three modes focus on how the driver processes incoming characters before delivering them to an application. Figure 15.1 summarizes the three modes and gives their characteristics.

| Mode | Meaning |
|---|---|
| raw | The driver delivers each incoming character as it arrives without echoing the character, buffering a line of text, performing translation, or controlling the output flow |
| cooked | The driver buffers input, echoes characters in a readable form, honors backspace and line kill, allows type-ahead, handles flow control, and delivers an entire line of text |
| cbreak | The driver handles character translation, echoing, and flow control, but instead of buffering an entire line of text, the driver delivers each incoming characters as it arrives |

**Figure 15.1** Three modes supported by the tty abstraction.

*Cooked mode* is intended to handle interactive keyboard input. Each time it receives a character, the driver echoes the character (i.e., transmits a copy of the character to the output), which allows a user to see characters as they are typed. Echo is not mandatory. Instead, the driver has a parameter to control character echoing, which means an application can turn off echo to prompt for a password. Cooked mode supports *line buffering*, which means that the driver collects all characters of a line before delivering them to a reading process. Because the tty driver performs character echo and other functions at interrupt time, a user can type ahead, even if no application is reading characters (e.g., a user can type the next command while the current command is running). The chief advantage of line buffering arises from the ability to edit the line, either by backspacing or typing a special character that erases the entire line and allows the user to begin entering the line again.

Cooked mode provides two additional functions. First, it handles output *flow control*, allowing a user to temporarily stop and later restart output. When flow control is enabled, typing *control-s* stops output and typing *control-q* restarts output. Second, cooked mode handles input mapping. In particular, some computers or applications use a two-character sequence of *carriage return* (*cr*) and *linefeed* (*lf*) to terminate a line of

text, and others use only a single character. Cooked mode contains a *crlf*† parameter that controls how the driver handles line termination. When a user presses the key labeled *ENTER* or *RETURN*, the driver consults the parameter to decide whether to pass a *linefeed* (also called a *NEWLINE*) character to the application or to map the *linefeed* into a pair of characters, *carriage return* followed by *linefeed*.

*Raw mode* is intended to give applications access to input characters with no preprocessing. In raw mode, the tty driver merely delivers input without interpreting or changing characters. The driver does not echo characters nor does it handle flow control. Raw mode is useful when handling non-interactive communication, such as downloading a binary file over a serial line or using a serial device to control a sensor.

*Cbreak mode* provides a compromise between cooked and raw modes. In cbreak mode, each character is delivered to the application instantly, without waiting to accumulate a line of text. Thus, the driver does not buffer input, nor does the driver support backspace or line kill functions. However, the driver does handle both character echo and flow control.

## 15.3 Organization Of A Tty Device Driver

Like most device drivers, the example tty driver is partitioned into an *upper half* that contains functions called by application processes (indirectly through the device switch table), and a *lower half* that contains functions invoked when the device interrupts. The two halves share a data structure that contains information about the device, the current mode of the driver, and buffers for incoming and outgoing data. In general, upper-half functions move data to or from the shared structure and have minimal interaction with the device hardware. For example, an upper-half function places outgoing data in the shared structure where a lower-half function can access and send the data to the device. Similarly, the lower half places incoming data in the shared structure where an upper-half function can extract it.

The motivation for driver partitioning can be difficult to appreciate at first. We will see, however, that dividing a driver into two halves is fundamental because the division allows a system designer to decouple normal processing from hardware interrupt processing and understand exactly how each function is invoked. The point is:

> *When creating a device driver, a programmer must be careful to preserve the division between upper-half and lower-half functions because upper-half functions are called by application processes and lower-half functions are invoked by interrupts.*

†Pronounced *curl-if*.

## 15.4 Request Queues And Buffers

The shared data structure in a driver usually contains two key items:

- A queue of requests
- Input and output buffers

*Request queue*. In principle, the most important item found in the data structure shared by upper-half and lower-half functions is a queue into which the upper half places requests. Conceptually, the *request queue* connects high-level operations that applications specify and low-level actions that must be performed on the device. Each driver has its own set of requests, and the contents of elements on a request queue depend on the underlying device as well as the operations to be performed. For example, requests issued to a disk device specify the direction of transfer (read or write), a location on the disk, and the amount of data to be transferred. Requests issued to a network device might specify a set of packets to transmit over the network. Our example driver does not need a separate request queue because the only operations are to transmit an outgoing character or receive an incoming character. Thus, the queue of outgoing characters serves as a transmission request, and space in the queue of incoming characters serves as a reception request.

*Buffers*. A driver uses an *output buffer* to hold data being sent to the device. An outgoing data item remains in the buffer from the time an application sends the item until the device is ready to accept it. An *input buffer* holds data that has arrived from a device, and an incoming item remains in the buffer from the time the device deposits the item until a process requests it.

Buffers are important for several reasons. First, a driver can accept incoming data and place it in an input buffer before a user process reads the data. Input buffering is especially important for devices like a network interface or a keyboard because packets can arrive at any time and a user can strike a key at any time. Second, for a device such as a disk that transfers data in blocks, the operating system must obtain an entire block, even if an application only reads one character. By placing the block in a buffer, the system can satisfy subsequent requests from the buffer. Third, buffering permits the driver to perform I/O concurrently with processing. When a process writes data, the driver copies the data into an output buffer, starts an output operation, and allows the process to continue executing.

The example tty driver uses three circular character buffers for each serial device: one for input, one for output, and one for echoed characters. Echoed characters are kept in a buffer separate from normal output because echoed characters have higher priority. We think of each buffer as a conceptual queue, with characters being inserted at the tail and removed from the head. Figure 15.2 illustrates the concept of a circular output buffer, and shows the implementation with an array of bytes in memory.

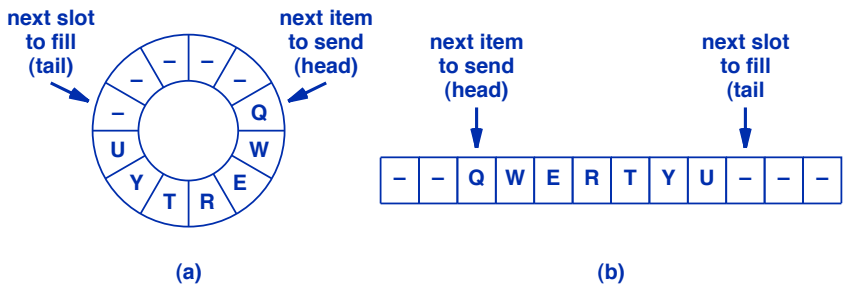



**Figure 15.2** (a) A circular output buffer acting as a queue, and (b) the implementation with an array of bytes.

Output functions deposit characters to be sent in the output buffer and return to their caller. When it places characters in the output buffer, an upper-half function must also start output interrupts on the device. Whenever the device generates an output interrupt, the lower half extracts up to sixteen characters from the output buffer, and deposits the characters in the device's output FIFO. Once all characters have been transmitted, the device will interrupt again. Thus, output continues until the output buffer becomes empty at which time the driver stops output and the device becomes idle.

Input works the other way around. Whenever it receives characters, the device interrupts and the interrupt dispatcher calls a lower-half function (i.e., *ttyInterrupt*). The interrupt handler extracts the characters from the device FIFO and deposits them in the circular input buffer. When a process calls an upper-half function to read input, the upper-half function extracts characters from the input buffer.

Conceptually, the two halves of a driver only communicate through shared buffers. Upper-half functions place outgoing data in a buffer and extract incoming data from a buffer. The lower half extracts outgoing data from the buffer and sends it to the device, and places incoming data in the buffer. To summarize:

*Upper-half functions transfer data between processes and buffers; the lower half transfers data between buffers and the device hardware.*

## 15.5 Synchronization Of Upper Half And Lower Half

In practice, the two halves of the driver usually need to do more than manipulate a shared data structure. For example, an upper-half function may need to start an output transfer if a device is idle. More important, the two halves need to coordinate operations on the request queue and the buffers. For example, if all slots in the output buffer are full when a process tries to write data, the process must be blocked. Later, when characters have been sent to the device and buffer space becomes available, the blocked process must be allowed to proceed. Similarly, if the input buffer is empty when a pro-

cess attempts to read from a device, the process must be blocked. Later, when input has been received and placed in the buffer, the process that is waiting for input must be allowed to proceed.

At first glance, synchronization between the upper half and lower half of a driver appears to consist of two instances of *producer–consumer coordination* that can be solved easily with semaphores. On output, the upper-half functions produce data that the lower-half functions consume, and on input, the lower half produces input data that the upper-half functions consume. Input poses no problem for the producer–consumer paradigm; a semaphore can be created that handles coordination. When a process calls an upper-half input function, the process *waits* on the input semaphore until the lower half produces an input data item and *signals* the semaphore.

Output poses an added twist. To understand the problem, recall our restriction on interrupt processing: because it can be executed by the null process, an interrupt routine cannot call a function that moves the executing process to any state other than *ready* or *current*. In particular, lower-half routines cannot call *wait*. Consequently, a driver cannot be designed in which a semaphore allows upper-half functions to produce data and lower-half functions to consume data.

How can upper– and lower–half functions coordinate to control output? Surprisingly, a semaphore solves the problem easily. The trick is to turn around the call to *wait* by changing the purpose of the output semaphore. Instead of having a lower-half routine *wait* for the upper half to produce data, we arrange for the upper half to *wait* for space in the buffer. Thus, we do not view the lower half as a consumer. Instead, a lower-half output function acts as a producer to generate space (i.e., slots) in the buffer, and signals the output semaphore for each slot. To summarize:

> *Semaphores can be used to coordinate the upper half and lower half of a device driver. To avoid having lower-half functions block, output is handled by arranging for upper-half functions to wait for buffer space.*

## 15.6 Hardware Buffers And Driver Design

The design of the hardware can complicate driver design. For example, consider the *Universal Asynchronous Transmitter and Receiver* hardware in the E2100L. The device contains two onboard buffers, known as *FIFOs*. One FIFO handles incoming characters, and the other handles outgoing characters. Each FIFO holds sixteen characters. The device does not interrupt each time a character arrives. Instead, the hardware generates an interrupt when the first character arrives, but continues to add characters to the input FIFO if they arrive before the interrupt has been serviced. Thus, when it receives an input interrupt, the driver must repeatedly extract characters from the FIFO until the FIFO is empty.

How do multiple input characters affect the driver design? Consider the case where a process is blocked on an input semaphore, waiting for a character to arrive. In theory, once it extracts a character from the device and places the character in the input buffer, the driver should signal the semaphore to reschedule to indicate that a character is available. However, doing so may cause an immediate context switch, leaving additional characters in the FIFO unprocessed. To avoid the problem, our driver uses *sched_cntl*† to defer rescheduling temporarily. After all characters have been extracted from the input FIFO and processed, the driver again calls *sched_cntl* to permit other processes to run.

## 15.7 Tty Control Block And Data Declarations

Recall that if a system contains multiple copies of a given hardware device, the operating system keeps one copy of the device driver code, but creates a separate shared data structure for each device. Some systems use the term *control block* to describe the shared data structure, and say that one control block is allocated *per physical device*. When it runs, a device driver function receives an argument that identifies the control block to use. Thus, if a particular system has three serial devices that use the tty abstraction, the operating system contains only one copy of the functions that read and write to a tty device, but contains three separate copies of a tty control block.

A control block stores information about the device, the driver, and the request queue. The control block either contains buffers or contains pointers to buffers in memory‡. Control blocks also store information that the upper half and lower half use to coordinate. For example, because the example tty driver uses a semaphore to coordinate access to the output buffer and a semaphore to coordinate access to the input buffer, the tty control block stores the two semaphore IDs.

The code in file *tty.h* contains the declaration of the tty control block structure, which is named *ttycblk*:

†The code for *sched_cntl* can be found on page 84.

‡On some systems, I/O buffers must be placed in a special region of memory to permit devices to access the buffers directly.

```
/* tty.h */

#define TY_OBMINSP              20      /* min space in buffer before   */
                                        /* processes awakened to write  */
#define TY_EBUFLEN              20      /* size of echo queue           */

/* Size constants */

#ifndef Ntty
#define Ntty            1               /* number of serial tty lines   */
#endif
#ifndef TY_IBUFLEN
#define TY_IBUFLEN      128             /* num. chars in input queue    */
#endif
#ifndef TY_OBUFLEN
#define TY_OBUFLEN      64              /* num. chars in output queue   */
#endif

/* Mode constants for input and output modes */

#define TY_IMRAW        'R'             /* raw mode => nothing done     */
#define TY_IMCOOKED     'C'             /* cooked mode => line editing  */
#define TY_IMCBREAK     'K'             /* honor echo, etc, no line edit*/
#define TY_OMRAW        'R'             /* raw mode => normal processing*/

struct  ttycblk {                       /* tty line control block       */
        char    *tyihead;               /* next input char to read      */
        char    *tyitail;               /* next slot for arriving char  */
        char    tyibuff[TY_IBUFLEN];    /* input buffer (holds one line)*/
        sid32   tyisem;                 /* input semaphore              */
        char    *tyohead;               /* next output char to xmit     */
        char    *tyotail;               /* next slot for outgoing char  */
        char    tyobuff[TY_OBUFLEN];    /* output buffer                */
        sid32   tyosem;                 /* output semaphore             */
        char    *tyehead;               /* next echo char to xmit       */
        char    *tyetail;               /* next slot to deposit echo ch */
        char    tyebuff[TY_EBUFLEN];    /* echo buffer                  */
        char    tyimode;                /* input mode raw/cbreak/cooked */
        bool8   tyiecho;                /* is input echoed?             */
        bool8   tyieback;               /* do erasing backspace on echo?*/
        bool8   tyevis;                 /* echo control chars as ^X ?   */
        bool8   tyecrlf;                /* echo CR-LF for newline?      */
        bool8   tyicrlf;                /* map '\r' to '\n' on input?   */
        bool8   tyierase;               /* honor erase character?       */
        char    tyierasec;              /* erase character (backspace)  */
```

```
        bool8   tyeof;                  /* honor EOF character?         */
        char    tyeofch;                /* EOF character (usually ^D)   */
        bool8   tyikill;                /* honor line kill character?   */
        char    tyikillc;               /* line kill character          */
        int32   tyicursor;              /* current cursor position      */
        bool8   tyoflow;                /* honor ostop/ostart?          */
        bool8   tyoheld;                /* output currently being held? */
        char    tyostop;                /* character that stops output  */
        char    tyostart;               /* character that starts output */
        bool8   tyocrlf;                /* output CR/LF for LF ?        */
        char    tyifullc;               /* char to send when input full */
};
extern  struct  ttycblk ttytab[];

/* Characters with meaning to the tty driver */

#define TY_BACKSP       '\b'            /* Backspace character          */
#define TY_BELL         '\07'           /* Character for audible beep   */
#define TY_EOFCH        '\04'           /* Control-D is EOF on input    */
#define TY_BLANK        ' '             /* Blank                        */
#define TY_NEWLINE      '\n'            /* Newline == line feed         */
#define TY_RETURN       '\r'            /* Carriage return character    */
#define TY_STOPCH       '\023'          /* Control-S stops output       */
#define TY_STRTCH       '\021'          /* Control-Q restarts output    */
#define TY_KILLCH       '\025'          /* Control-U is line kill       */
#define TY_UPARROW      '^'             /* Used for control chars (^X)  */
#define TY_FULLCH       TY_BELL         /* char to echo when buffer full*/

/* Tty control function codes */

#define TC_NEXTC        3               /* look ahead 1 character       */
#define TC_MODER        4               /* set input mode to raw        */
#define TC_MODEC        5               /* set input mode to cooked     */
#define TC_MODEK        6               /* set input mode to cbreak     */
#define TC_ICHARS       8               /* return number of input chars */
#define TC_ECHO         9               /* turn on echo                 */
#define TC_NOECHO       10              /* turn off echo                */
```

The key components of the *ttycblk* structure consist of the input buffer, *tyibuff*, an output buffer, *tyobuff*, and a separate echo buffer, *tyebuff*. Each buffer used in the tty driver is implemented as an array of characters. The driver treats each buffer as a circular list, with location zero in an array treated as if it follows the last location. Head and tail pointers give the address of the next location in the array to fill, and the next loca-

tion in the array to empty, respectively. Thus, a programmer can remember an easy rule:

> *A character is always inserted at the tail and taken from the head, independent of whether a buffer is used for input or output.*

Initially, the head and tail each point to location zero, but there is never any confusion about whether an input or output buffer is completely empty or completely full because each buffer has a semaphore that gives the count of characters in the buffer. Semaphore *tyisem* controls the input buffer, and a non-negative count *n* means the buffer contains *n* characters. Semaphore *tyosem* controls the output buffer, and a non-negative count *n* means the buffer contains *n* unfilled slots. The echo buffer is an exception. Our design assumes echo is used for a human typing, which means that only a few characters will ever occupy the echo queue. Therefore, we assume that no overflow will occur, which means that no semaphore is needed to control the queue.

## 15.8 Minor Device Numbers

We said that the configuration program assigns each device in the system a unique device ID. It is important to know that although a system can contain multiple physical devices that use a given abstraction, the IDs assigned to the devices may not be contiguous values. Thus, if a system has three tty devices, the configuration program may assign them device IDs 2, 7, and 8.

We also said that the operating system must allocate one control block per device. For example, if a system contains three tty devices, the system must allocate three copies of the tty control block. Many systems employ a technique that permits efficient access of a control block for a given device. The system assigns a *minor device number* to each copy of the device, and chooses minor device numbers to be integers starting at zero. Thus, if a system contains three tty devices, they will be assigned minor device numbers 0, 1, and 2.

How does assigning minor device numbers sequentially make access efficient? A minor device number can be used as an index into an array of control blocks. For example, consider how tty control blocks are allocated. As file *tty.h* illustrates, the control blocks are placed in array *ttytab*. The system configuration program defines constant *Ntty* to be the number of tty devices, which is used to declare the size of array *ttytab*. The configuration program assigns each tty device a minor device number starting at 0 and ending at *Ntty*-1. The minor device number is placed in the device switch table entry. Both interrupt-driven routines in the lower half and driver routines in the upper half can access the minor device number and use it as an index into array *ttytab*.

## 15.9 Upper–Half Tty Character Input (ttyGetc)

Four functions, *ttyGetc*, *ttyPutc*, *ttyRead*, and *ttyWrite*, form the main foundation for the upper half of the tty driver. The functions correspond to the high-level operations *getc*, *putc*, *read*, and *write* described in Chapter 14. The simplest driver routine is *ttyGetc*. The code can be found in file *ttyGetc.c*.

```
/* ttyGetc.c - ttyGetc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyGetc - read one character from a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyGetc(
          struct dentry *devptr         /* entry in device switch table */
        )
{
        char    ch;
        struct  ttycblk *typtr;         /* pointer to ttytab entry      */

        typtr = &ttytab[devptr->dvminor];

        /* Wait for a character in the buffer */

        wait(typtr->tyisem);
        ch = *typtr->tyihead++;         /* extract one character        */

        /* Wrap around to beginning of buffer, if needed */

        if (typtr->tyihead >= &typtr->tyibuff[TY_IBUFLEN]) {
                typtr->tyihead = typtr->tyibuff;
        }

        if ( (typtr->tyimode == TY_IMCOOKED) && (typtr->tyeof) &&
             (ch == typtr->tyeofch) ) {
                return (devcall)EOF;
        }

        return (devcall)ch;
}
```

When called, *ttyGetc* first retrieves the minor device number from the device switch table, and uses it as an index into array *ttytab* to locate the correct control block. It then executes *wait* on the input semaphore, *tyisem*, blocking until the lower half has deposited a character in the buffer. When *wait* returns, *ttyGetc* extracts the next character from the input buffer and updates the head pointer to make it ready for subsequent extractions. Normally, *ttyGetc* returns the character to its caller. However, one special case arises: if the driver is honoring *end-of-file* and the character matches the end-of-file character (field *tyeofch* in the control block), *ttyGetc* returns constant *EOF*.

## 15.10 Generalized Upper–Half Tty Input (ttyRead)

The *read* operation can be used to obtain multiple characters in a single operation. The driver function that implements the *read* operation, *ttyRead*, is shown below in file *ttyRead.c*. *TtyRead* is conceptually straightforward: it calls *ttyGetc* repeatedly to obtain characters. When the driver is operating in cooked mode, *ttyRead* returns a single line of input, stopping after a *NEWLINE* or *RETURN* character; when operating in other modes, *ttyRead* reads characters without testing for an end-of-line.

```
/* ttyRead.c - ttyRead */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyRead - read character(s) from a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyRead(
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer of characters         */
          int32 count                   /* count of character to read   */
        )
{
        struct  ttycblk *typtr;         /* pointer to tty control block */
        int32   avail;                  /* characters available in buff.*/
        int32   nread;                  /* number of characters read    */
        int32   firstch;                /* first input character on line*/
        char    ch;                     /* next input character         */

        if (count < 0) {
                return SYSERR;
        }
        typtr= &ttytab[devptr->dvminor];

        if (typtr->tyimode != TY_IMCOOKED) {
```

```
                /* For count of zero, return all available characters */

                if (count == 0) {
                        avail = semcount(typtr->tyisem);
                        if (avail == 0) {
                                return 0;
                        } else {
                                count = avail;
                        }
                }
                for (nread = 0; nread < count; nread++) {
                        *buff++ = (char) ttyGetc(devptr);
                }
                return nread;
        }

        /* Block until input arrives */

        firstch = ttyGetc(devptr);

        /* Check for End-Of-File */

        if (firstch == EOF) {
                return (EOF);
        }

        /* read up to a line */

        ch = (char) firstch;
        *buff++ = ch;
        nread = 1;
        while ( (nread < count) && (ch != TY_NEWLINE) &&
                        (ch != TY_RETURN) ) {
                ch = ttyGetc(devptr);
                *buff++ = ch;
                nread++;
        }
        return nread;
}
```

The semantics of how *read* operates on terminals illustrates how the I/O primitives can be adapted to a variety of devices and modes. For example, an application that uses raw mode may need to read all the characters available from the input buffer without blocking. *TtyRead* cannot simply call *ttyGetc* repeatedly because *ttyGetc* will block

once the buffer is empty. To accommodate non-blocking requests, our driver allows what might otherwise be considered an illegal operation: it interprets a request to *read* zero characters as a request to "read all characters that are waiting."

The code in *ttyRead* shows how the zero length requests are handled in raw mode: the driver uses *semcount* to obtain the current count of the input semaphore, *tyisem*. It then knows exactly how many calls can be made to *ttyGetc* without blocking.

For cooked mode, the driver blocks until at least one character arrives. It handles the special case of an end-of-file character, and then calls *ttyGetc* repeatedly to read the rest of the line.

## 15.11 Upper–Half Tty Character Output (ttyPutc)

The upper-half output routines are almost as simple as the upper-half input routines. *TtyPutc* waits for space in the output buffer, deposits the specific character in the output queue, *tyobuff*, and increments the tail pointer, *tyotail*. File *ttyPutc.c* contains the code.

```
/* ttyPutc.c - ttyPutc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyPutc - write one character to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyPutc(
        struct  dentry  *devptr,        /* entry in device switch table */
        char    ch                      /* character to write           */
        )
{
        struct  ttycblk *typtr;         /* pointer to tty control block */

        typtr = &ttytab[devptr->dvminor];

        /* Handle output CRLF by sending CR first */

        if ( ch==TY_NEWLINE && typtr->tyocrlf ) {
                ttyPutc(devptr, TY_RETURN);
        }

        wait(typtr->tyosem);            /* wait for space in queue */
        *typtr->tyotail++ = ch;

        /* Wrap around to beginning of buffer, if needed */
```

```
        if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
                typtr->tyotail = typtr->tyobuff;
        }

        /* Start otuput in case device is idle */

        ttyKickOut(typtr, (struct uart_csreg *)devptr->dvcsr);

        return OK;
}
```

In addition to the processing mentioned above, *ttyPutc* honors one of the tty parameters, *tyocrlf*, and starts output. When *tyocrlf* is TRUE, each *NEWLINE* should map to the combination *RETURN* plus *NEWLINE*. To write the *RETURN* character, *ttyPutc* calls itself recursively.

## 15.12 Starting Output (ttyKickOut)

Just before it returns, *ttyPutc* calls *ttyKickOut* to start output. In fact, *ttyKickOut* does not perform any output to the device because all output is handled by the lower-half function when an output interrupt occurs. To understand how *ttyKickOut* works, it is necessary to know how an operating system interacts with the device hardware. It may seem that when a character becomes ready for output, *ttyPutc* would take the following steps:

```
Interact with the device to determine whether the device is busy;
if (the device is not busy) {
        send the character to the device;
} else {
        instruct the device to interrupt when output finishes;
}
```

Unfortunately, a device operates in parallel with the processor. Therefore, between the time the processor obtains the status and the time it instructs the device to interrupt, the device can finish.

To avoid a race condition, device hardware allows the operating system to request an interrupt without testing the device. Making a request is trivial: the driver merely needs to set a bit in one of the device control registers. The point is that no race condition occurs because setting the bit causes an interrupt whether the device is currently sending characters or idle. If the device is busy, the hardware waits until output finishes and the on-board buffer is empty before generating an interrupt; if the device is currently idle, the device interrupts immediately.

Setting the interrupt bit in the device only requires a single assignment statement; the code can be found in file *ttyKickOut.c*:

```
/* ttyKickOut.c - ttyKickOut */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyKickOut - "kick" the hardware for a tty device, causing it to
 *               generate an output interrupt (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyKickOut(
         struct ttycblk *typtr,         /* ptr to ttytab entry          */
         struct uart_csreg *uptr        /* address of UART's CSRs       */
        )
{
        /* Set output interrupts on the UART, which causes */
        /*   the device to generate an output interrupt    */

        uptr->ier = UART_IER_ERBFI | UART_IER_ETBEI | UART_IER_ELSI;

        return;
}
```

## 15.13 Upper–Half Tty Multiple Character Output (ttyWrite)

The *tty* driver also supports multiple-byte output transfers (i.e., *write*s). Driver function *ttyWrite*, found in file *ttyWrite.c*, handles the output of one or more bytes. *TtyWrite* begins by checking the argument *count*, which specifies the number of bytes to write. A negative count is invalid, and a count of zero is allowed, but means no characters are written.

Once it has finished checking argument *count*, *ttyWrite* enters a loop. On each iteration through the loop, *ttyWrite* extracts the next character from the user's buffer and calls *ttyPutc* to send the character to the output buffer. As we have seen, *ttyPutc* will proceed until the output buffer is full, at which time the call to *ttyPutc* will block until space becomes available.

```
/* ttyWrite.c - ttyWrite, writcopy */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyWrite - write character(s) to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyWrite(
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer of characters         */
          int32 count                   /* count of character to write  */
        )
{

        if (count < 0) {
                return SYSERR;
        } else if (count == 0){
                return OK;
        }

        for (; count>0 ; count--) {
                ttyPutc(devptr, *buff++);
        }
        return OK;
}
```

## 15.14 Lower–Half Tty Driver Function (ttyInterrupt)

The lower half of the tty driver is invoked when an interrupt occurs. It consists of function *ttyInterrupt*, shown below in file *ttyInterrupt.c*:

```
/* ttyInterrupt.c - ttyInterrupt */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyInterrupt - handle an interrupt for a tty (serial) device
 *------------------------------------------------------------------------
 */
interrupt ttyInterrupt(void)
{
        struct  dentry  *devptr;        /* pointer to devtab entry      */
        struct  ttycblk *typtr;         /* pointer to ttytab entry      */
        struct  uart_csreg *uptr;       /* address of UART's CSRs       */
        int32   iir = 0;                /* interrupt identification     */
        int32   lsr = 0;                /* line status                  */

        /* For now, the CONSOLE is the only serial device */

        devptr = (struct dentry *)&devtab[CONSOLE];

        /* Obtain the CSR address for the UART */

        uptr = (struct uart_csreg *)devptr->dvcsr;

        /* Obtain a pointer to the tty control block */

        typtr = &ttytab[ devptr->dvminor ];

        /* Decode hardware interrupt request from UART device */

        /* Check interrupt identification register */

        iir = uptr->iir;
        if (iir & UART_IIR_IRQ) {
                return;
        }

        /* Decode the interrupt cause based upon the value extracted    */
        /* from the UART interrupt identification register.  Clear      */
        /* the interrupt source and perform the appropriate handling    */
        /* to coordinate with the upper half of the driver              */

        iir &= UART_IIR_IDMASK;         /* Mask off the interrupt ID */
        switch (iir) {
```

```
            /* Receiver line status interrupt (error) */

            case UART_IIR_RLSI:
                lsr = uptr->lsr;
                return;

            /* Receiver data available or timed out */

            case UART_IIR_RDA:
            case UART_IIR_RTO:

                sched_cntl(DEFER_START);

                /* For each char in UART buffer, call ttyInter_in */

                while (uptr->lsr & UART_LSR_DR) { /* while chars avail */
                        ttyInter_in(typtr, uptr);
                }

                sched_cntl(DEFER_STOP);

                return;

            /* Transmitter output FIFO is empty (i.e., ready for more)  */

            case UART_IIR_THRE:
                lsr = uptr->lsr;  /* Read from LSR to clear interrupt */
                ttyInter_out(typtr, uptr);
                return;

            /* Modem status change (simply ignore) */

            case UART_IIR_MSC:
                return;
            }
}
```

Recall that a handler is invoked indirectly — the interrupt dispatcher calls the handler whenever the device interrupts. We will see that the tty initialization routine arranges the connection between the dispatcher and *ttyInterrupt*. For now, it is sufficient to know that the handler will be invoked whenever: the device has received (one or more) incoming characters, or the device has sent all the characters in its output FIFO and is ready for more.

After obtaining the device's CSR address from the device switch table, *ttyInterrupt* loads the device CSR address into *uptr*, and then uses *uptr* to access the device. The key step consists of reading the interrupt identification register and using the value to determine the exact reason for the interrupt. The two reasons of interest are an input interrupt (data has arrived) or an output interrupt (i.e., the transmitter FIFO is empty and the driver can send additional characters).

## 15.15 Output Interrupt Processing (ttyInter_out)

Output interrupt processing is the easiest to understand. When an output interrupt occurs, the device has transmitted all characters from the onboard FIFO and is ready for more. *TtyInterrupt* clears the interrupt and calls *ttyInter_out* to restart output. The code for *ttyInter_out* can be found in file *ttyInter_out.c*

```
/* ttyInter_out.c - ttyInter_out */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyInter_out - handle an output on a tty device by sending more
 *                 characters to the device FIFO (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyInter_out(
         struct ttycblk *typtr,         /* ptr to ttytab entry          */
         struct uart_csreg *uptr        /* address of UART's CSRs       */
        )
{

        int32   ochars;                 /* number of output chars sent  */
                                        /*   to the UART                */
        int32   avail;                  /* available chars in output buf*/
        int32   uspace;                 /* space left in onboard UART   */
                                        /*   output FIFO                */

        /* If output is currently held, turn off output interrupts */

        if (typtr->tyoheld) {
                uptr->ier = UART_IER_ERBFI | UART_IER_ELSI;
                return;
        }

        /* If echo and output queues empty, turn off output interrupts */
```

```
        if ( (typtr->tyehead == typtr->tyetail) &&
             (semcount(typtr->tyosem) >= TY_OBUFLEN) ) {
                uptr->ier = UART_IER_ERBFI | UART_IER_ELSI;
                return;
        }

        /* Initialize uspace to the size of the transmit FIFO */

        uspace = UART_FIFO_SIZE;

        /* While onboard FIFO is not full and the echo queue is */
        /* nonempty, xmit chars from the echo queue             */

        while ( (uspace>0) &&  typtr->tyehead != typtr->tyetail) {
                uptr->buffer = *typtr->tyehead++;
                if (typtr->tyehead >= &typtr->tyebuff[TY_EBUFLEN]) {
                        typtr->tyehead = typtr->tyebuff;
                }
                uspace--;
        }

        /* While onboard FIFO is not full and the output queue  */
        /* is nonempty, xmit chars from the output queue        */

        ochars = 0;
        avail = TY_OBUFLEN - semcount(typtr->tyosem);
        while ( (uspace>0) &&  (avail > 0) ) {
                uptr->buffer = *typtr->tyohead++;
                if (typtr->tyohead >= &typtr->tyobuff[TY_OBUFLEN]) {
                        typtr->tyohead = typtr->tyobuff;
                }
                avail--;
                uspace--;
                ochars++;
        }
        if (ochars > 0) {
                signaln(typtr->tyosem, ochars);
        }
        return;
}
```

*TtyInter_out* makes a series of tests before starting output. For example, output should not be started if a user has entered control-S. Similarly, there is no need to start output if both the echo and output queues are empty. To understand how *ttyInter_out* starts output, recall that the underlying hardware has an onboard FIFO that can hold multiple outgoing characters. Once it has determined that output should proceed,

*ttyInter_out* can send up to *UART_FIFO_SIZE* (16) characters to the device. Characters are sent until the FIFO is full or the buffers are empty, whichever occurs first. The echo queue has highest priority. Therefore, *ttyInter_out* first sends characters from the echo queue. If slots remain, *ttyInter_out* sends characters from the output queue.

Conceptually, each time it removes a character from the output queue and sends the character to the device, *ttyInter_out* should signal the output semaphore to indicate that another space is available in the buffer. However, because a call to *signal* can reschedule, *ttyInter_out* does not call *signal* immediately. Instead, it merely increments variable *ochars* to count the number of additional slots being created in the output queue. Once it has filled the FIFO (or has emptied the output queue), *ttyInter_out* calls *signaln* to indicate that space is available in the buffer.

## 15.16 Tty Input Processing (ttyInter_in)

Input interrupt processing is more complex than output processing because the onboard input FIFO can contain more than one character. Thus, to handle an input interrupt, *ttyInterrupt*† enters a loop: while the onboard FIFO is not empty, *ttyInterrupt* calls *ttyInter_in*, which extracts and processes one character from the UART's input FIFO. To prevent rescheduling until the loop completes and all characters have been extracted from the device, *ttyInterrupt* uses *sched_cntl*. Thus, although *ttyInter_in* calls *signal* to make each character available, no rescheduling occurs until all available characters have been extracted from the device.

Processing individual input characters is the most complex part of the tty device driver because it includes code for details such as character echo and line editing. Function *ttyInter_in* handles the processing for *raw*, *cbreak*, and *cooked* modes. File *ttyInter_in.c* contains the code.

```
/* ttyInter_in.c ttyInter_in, erase1, eputc, echoch */

#include <xinu.h>

local   void    erase1(struct ttycblk *, struct uart_csreg *);
local   void    echoch(char, struct ttycblk *, struct uart_csreg *);
local   void    eputc(char, struct ttycblk *, struct uart_csreg *);

/*------------------------------------------------------------------------
 *  ttyInter_in  --  handle one arriving char (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyInter_in (
          struct ttycblk *typtr,        /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
```

†The code for *ttyInterrupt* can be found on page 284.

```
{
        char    ch;                     /* next char from device        */
        int32   avail;                  /* chars available in buffer    */

        ch = uptr->buffer;              /* extract char. from device    */

        /* Compute chars available */

        avail = semcount(typtr->tyisem);
        if (avail < 0) {                /* one or more processes waiting*/
                avail = 0;
        }

        /* Handle raw mode */

        if (typtr->tyimode == TY_IMRAW) {
                if (avail >= TY_IBUFLEN) { /* no space => ignore input  */
                        return;
                }

                /* Place char in buffer with no editing */

                *typtr->tyitail++ = ch;

                /* Wrap buffer pointer  */

                if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
                        typtr->tyotail = typtr->tyobuff;
                }

                /* Signal input semaphore and return */

                signal(typtr->tyisem);
                return;
        }

        /* Handle cooked and cbreak modes (common part) */

        if ( (ch == TY_RETURN) && typtr->tyicrlf ) {
                ch = TY_NEWLINE;
        }

        /* If flow control is in effect, handle ^S and ^Q */

        if (typtr->tyoflow) {
```

```
        if (ch == typtr->tyostart) {        /* ^Q starts output */
                typtr->tyoheld = FALSE;
                ttyKickOut(typtr, uptr);
                return;
        } else if (ch == typtr->tyostop) {  /* ^S stops output  */
                typtr->tyoheld = TRUE;
                return;
        }
}

typtr->tyoheld = FALSE;         /* Any other char starts output */

if (typtr->tyimode == TY_IMCBREAK) {       /* Just cbreak mode  */

        /* If input buffer is full, send bell to user */

        if (avail >= TY_IBUFLEN) {
                eputc(typtr->tyifullc, typtr, uptr);
        } else {        /* Input buffer has space for this char */
                *typtr->tyitail++ = ch;

                /* Wrap around buffer */

                if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
                        typtr->tyitail = typtr->tyibuff;
                }
                if (typtr->tyiecho) {   /* are we echoing chars?*/
                        echoch(ch, typtr, uptr);
                }
        }
        return;

} else {        /* Just cooked mode (see common code above)     */

        /* Line kill character arrives - kill entire line */

        if (ch == typtr->tyikillc && typtr->tyikill) {
                typtr->tyitail -= typtr->tyicursor;
                if (typtr->tyitail < typtr->tyibuff) {
                        typtr->tyihead += TY_IBUFLEN;
                }
                typtr->tyicursor = 0;
                eputc(TY_RETURN, typtr, uptr);
                eputc(TY_NEWLINE, typtr, uptr);
                return;
```

```
}

/* Erase (backspace) character */

if ( (ch == typtr->tyierasec) && typtr->tyierase) {
        if (typtr->tyicursor > 0) {
                typtr->tyicursor--;
                erase1(typtr, uptr);
        }
        return;
}

/* End of line */

if ( (ch == TY_NEWLINE) || (ch == TY_RETURN) ) {
        if (typtr->tyiecho) {
                echoch(ch, typtr, uptr);
        }
        *typtr->tyitail++ = ch;
        if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
                typtr->tyitail = typtr->tyibuff;
        }

        /*  Make entire line (plus \n or \r) available */

        signaln(typtr->tyisem, typtr->tyicursor + 1);
        typtr->tyicursor = 0;   /* Reset for next line  */
        return;
}

/* Character to be placed in buffer - send bell if      */
/*      buffer has overflowed                           */

avail = semcount(typtr->tyisem);
if (avail < 0) {
        avail = 0;
}
if ((avail + typtr->tyicursor) >= TY_IBUFLEN-1) {
        eputc(typtr->tyifullc, typtr, uptr);
        return;
}

/* EOF character: recognize at beginning of line, but   */
/*      print and ignore otherwise.                     */
```

```
                if (ch == typtr->tyeofch && typtr->tyeof) {
                        if (typtr->tyiecho) {
                                echoch(ch, typtr, uptr);
                        }
                        if (typtr->tyicursor != 0) {
                                return;
                        }
                        *typtr->tyitail++ = ch;
                        signal(typtr->tyisem);
                        return;
                }


                /* Echo the character */

                if (typtr->tyiecho) {
                        echoch(ch, typtr, uptr);
                }

                /* Insert character in the input buffer */

                typtr->tyicursor++;
                *typtr->tyitail++ = ch;

                /* Wrap around if needed */

                if (typtr->tyitail >= &typtr->tyibuff[TY_IBUFLEN]) {
                        typtr->tyitail = typtr->tyibuff;
                }
                return;
        }
}

/*------------------------------------------------------------------------
 *  erase1  --  erase one character honoring erasing backspace
 *------------------------------------------------------------------------
 */
local   void    erase1(
          struct ttycblk        *typtr, /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
{
        char    ch;                     /* character to erase           */

        if ( (--typtr->tyitail) < typtr->tyibuff) {
```

```
                typtr->tyitail += TY_IBUFLEN;
        }

        /* Pick up char to erase */

        ch = *typtr->tyitail;
        if (typtr->tyiecho) {                       /* Are we echoing?   */
                if (ch < TY_BLANK || ch == 0177) { /* Nonprintable      */
                        if (typtr->tyevis) {       /* Visual cntl chars */
                                eputc(TY_BACKSP, typtr, uptr);
                                if (typtr->tyieback) {  /* erase char   */
                                        eputc(TY_BLANK, typtr, uptr);
                                        eputc(TY_BACKSP, typtr, uptr);
                                }
                        }
                        eputc(TY_BACKSP, typtr, uptr);/* bypass up arrow*/
                        if (typtr->tyieback) {
                                eputc(TY_BLANK, typtr, uptr);
                                eputc(TY_BACKSP, typtr, uptr);
                        }
                } else {  /* A normal character that is printable       */
                        eputc(TY_BACKSP, typtr, uptr);
                        if (typtr->tyieback) {  /* erase the character  */
                                eputc(TY_BLANK, typtr, uptr);
                                eputc(TY_BACKSP, typtr, uptr);
                        }
                }
        }
        return;
}

/*-------------------------------------------------------------------------
 *  echoch  --  echo a character with visual and output crlf options
 *-------------------------------------------------------------------------
 */
local   void    echoch(
          char  ch,                     /* character to echo            */
          struct ttycblk *typtr,        /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
{
        if ((ch==TY_NEWLINE || ch==TY_RETURN) && typtr->tyecrlf) {
                eputc(TY_RETURN, typtr, uptr);
                eputc(TY_NEWLINE, typtr, uptr);
        } else if ( (ch<TY_BLANK||ch==0177) && typtr->tyevis) {
```

```
                eputc(TY_UPARROW, typtr, uptr); /* print ^x             */
                eputc(ch+0100, typtr, uptr);    /* make it printable    */
        } else {
                eputc(ch, typtr, uptr);
        }
}

/*-------------------------------------------------------------------------
 *  eputc - put one character in the echo queue
 *-------------------------------------------------------------------------
 */
local   void    eputc(
          char  ch,                     /* character to echo            */
          struct ttycblk *typtr,        /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
{
        *typtr->tyetail++ = ch;

        /* Wrap around buffer, if needed */

        if (typtr->tyetail >= &typtr->tyebuff[TY_EBUFLEN]) {
                typtr->tyetail = typtr->tyebuff;
        }
        ttyKickOut(typtr, uptr);
        return;
}
```

### 15.16.1 Raw Mode Processing

Raw mode is the easiest to understand, and accounts for only a few lines of code. In raw mode, *ttyInter_in* checks the input buffer to verify that space remains. To do so, it compares the count of the input semaphore (i.e., the number of characters that are available in the buffer) to the buffer size. If no space remains in the buffer, *ttyInter_in* merely returns (i.e., it discards the character). If space remains, *ttyInter_in* deposits the character at the tail of the input buffer, moves to the next buffer position, signals the input semaphore, and returns.

### 15.16.2 Cbreak Mode Processing

Cooked and cbreak mode share code that maps *RETURN* to *NEWLINE* and handles output flow control. Field *tyoflow* of the tty control block determines whether the driver currently honors flow control. If it does, the driver suspends output by setting *tyoheld* to *TRUE* when it receives character *tyostop*, and restarts output when it receives

character *tyostart*. Characters *tyostart* and *tyostop* are considered "control" characters, so the driver does not place them in the buffer.

Cbreak mode checks the input buffer, and sends character *tyifullc* if the buffer is full. Normally, *tyifullc* is a "bell" that causes the terminal to sound an audible alarm; the idea is that a human who is typing characters will hear the alarm and stop typing until characters have been read and more buffer space becomes available. If the buffer is not full, the code places the character in the buffer, and wraps around the pointer, if necessary. Finally, cbreak mode calls *echoch* to perform character echo.

### 15.16.3 Cooked Mode Processing

Cooked mode operates like cbreak mode except that it also performs line editing. The driver accumulates lines in the input buffer, using variable *tyicursor* to keep a count of the characters on the "current" line. When the erase character, *tyierasec*, arrives, *ttyInter_in* decrements *tyicursor* by one, backing up over the previous character, and calling function *erase1* to erase the character from the display. When the line kill character, *tyikillc*, arrives, *ttyInter_in* eliminates the current line by setting *tyicursor* to zero and moving the tail pointer back to the beginning of the line. Finally, when a *NEWLINE* or *RETURN* character arrives, *ttyInter_in* calls *signaln* to make the entire input line available. It resets *tyicursor* to zero for the next line. Note that the test for buffer full always leaves one extra space in the buffer for the end-of-line character (i.e., *NEWLINE*).

## 15.17 Tty Control Block Initialization (ttyInit)

Function *ttyInit*, shown below in file *ttyInit.c*, initializes fields in the tty control block. *TtyInit* uses *dvirq* as an index into the interrupt vector array, and assigns the vector the address of the interrupt function. *TtyInit* then initializes the control block to cooked mode, creates the input and output semaphores, and sets the buffer head and tail pointers. After driver parameters, buffers, and interrupt vectors have been initialized, *ttyInit* clears the receiver buffer in the hardware, enables receiver interrupts, and disables transmitter interrupts.

*TtyInit* initializes a tty to cooked mode, assuming it connects to a keyboard and display that a human will use. The parameters chosen work best for a video device that can backspace over characters on the display and erase them rather than a device that uses paper. In particular, the parameter *tyieback* causes *ttyInter_in* to echo three characters, backspace-space-backspace, when it receives the erase character, *tyierasec*. On a display screen, sending the three-character sequence gives the effect of erasing characters as the user backs over them. If you look again at *ttyInter_in*,† you will see that it carefully backs up the correct number of spaces, even if the user erases a control character that has been displayed as two printable characters.

†The code for *ttyInter_in* can be found on page 288.

```
/* ttyInit.c - ttyInit */

#include <xinu.h>

struct  ttycblk ttytab[Ntty];

/*------------------------------------------------------------------------
 *  ttyInit - initialize buffers and modes for a tty line
 *------------------------------------------------------------------------
 */
devcall ttyInit(
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  ttycblk *typtr;         /* pointer to ttytab entry      */
        struct  uart_csreg *uptr;       /* address of UART's CSRs       */

        typtr = &ttytab[ devptr->dvminor ];

        /* Initialize values in the tty control block */

        typtr->tyihead = typtr->tyitail =       /* set up input queue   */
                &typtr->tyibuff[0];             /*    as empty          */
        typtr->tyisem = semcreate(0);           /* input semaphore      */
        typtr->tyohead = typtr->tyotail =       /* set up output queue  */
                &typtr->tyobuff[0];             /*    as empty          */
        typtr->tyosem = semcreate(TY_OBUFLEN);  /* output semaphore     */
        typtr->tyehead = typtr->tyetail =       /* set up echo queue    */
                &typtr->tyebuff[0];             /*    as empty          */
        typtr->tyimode = TY_IMCOOKED;           /* start in cooked mode */
        typtr->tyiecho = TRUE;                  /* echo console input   */
        typtr->tyieback = TRUE;                 /* honor erasing bksp   */
        typtr->tyevis = TRUE;                   /* visual control chars */
        typtr->tyecrlf = TRUE;                  /* echo CRLF for NEWLINE*/
        typtr->tyicrlf = TRUE;                  /* map CR to NEWLINE    */
        typtr->tyierase = TRUE;                 /* do erasing backspace */
        typtr->tyierasec = TY_BACKSP;           /* erase char is ^H     */
        typtr->tyeof = TRUE;                    /* honor eof on input   */
        typtr->tyeofch = TY_EOFCH;              /* end-of-file character*/
        typtr->tyikill = TRUE;                  /* allow line kill      */
        typtr->tyikillc = TY_KILLCH;            /* set line kill to ^U  */
        typtr->tyicursor = 0;                   /* start of input line  */
        typtr->tyoflow = TRUE;                  /* handle flow control  */
        typtr->tyoheld = FALSE;                 /* output not held      */
        typtr->tyostop = TY_STOPCH;             /* stop char is ^S      */
```

```
        typtr->tyostart = TY_STRTCH;            /* start char is ^Q     */
        typtr->tyocrlf = TRUE;                  /* send CRLF for NEWLINE*/
        typtr->tyifullc = TY_FULLCH;            /* send ^G when buffer  */
                                                /*   is full            */

        /* Initialize the UART */

        uptr = (struct uart_csreg *)devtab[CONSOLE].dvcsr;

        /* Set baud rate */

        uptr->lcr = UART_LCR_8N1;       /* 8 bit char, No Parity, 1 Stop*/
        uptr->fcr = 0x00;               /* Disable FIFO for now         */
        /* OUT2 value is used to control the onboard interrupt tri-state*/
        /* buffer. It should be set high to generate interrupts         */
        uptr->mcr = UART_MCR_OUT2;      /* Turn on user-defined OUT2    */

        /* Enable interrupts */

        /* Enable UART FIFOs, clear and set interrupt trigger level     */
        uptr->fcr = UART_FCR_EFIFO | UART_FCR_RRESET
                        | UART_FCR_TRESET | UART_FCR_TRIG2;

        /* Register the interrupt handler for the dispatcher */

        interruptVector[devptr->dvirq] = (void *)devptr->dvintr;

        /* Ready to enable interrupts on the UART hardware */

        enable_irq(devptr->dvirq);

        ttyKickOut(typtr, uptr);

        return OK;
}
```

## 15.18 Device Driver Control

So far we have discussed driver functions that handle upper-half data transfer operations (e.g., *read* and *write*), functions that handle lower-half input and output interrupts, and an initialization function that sets parameters at system startup. The I/O interface defined in Chapter 14 provides another type of non-transfer function: *control*. Basically, *control* allows an application to control the device driver or the underlying device. In our example driver, function *ttyControl*, found in file *ttyControl.c*, provides basic control functions:

```
/* ttyControl.c - ttyControl */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyControl  -  control a tty device by setting modes
 *------------------------------------------------------------------------
 */
devcall ttyControl(
          struct dentry *devptr,        /* entry in device switch table */
          int32  func,                  /* function to perform          */
          int32  arg1,                  /* argument 1 for request       */
          int32  arg2                   /* argument 2 for request       */
        )
{
        struct  ttycblk *typtr;         /* pointer to tty control block */
        char    ch;                     /* character for lookahead      */

        typtr = &ttytab[devptr->dvminor];

        /* Process the request */

        switch ( func ) {

        case TC_NEXTC:
                wait(typtr->tyisem);
                ch = *typtr->tyitail;
                signal(typtr->tyisem);
                return (devcall)ch;

        case TC_MODER:
                typtr->tyimode = TY_IMRAW;
                return (devcall)OK;

        case TC_MODEC:
                typtr->tyimode = TY_IMCOOKED;
                return (devcall)OK;

        case TC_MODEK:
                typtr->tyimode = TY_IMCBREAK;
                return (devcall)OK;

        case TC_ICHARS:
                return(semcount(typtr->tyisem));
```

```
        case TC_ECHO:
                typtr->tyiecho = TRUE;
                return (devcall)OK;

        case TC_NOECHO:
                typtr->tyiecho = FALSE;
                return (devcall)OK;

        default:
                return (devcall)SYSERR;
        }
}
```

Function *TC_NEXTC* allows an application to "lookahead" (i.e., find out which character is the next one waiting to be read without actually reading the character). Three of the control functions (*TC_MODER*, *TC_MODEC*, and *TC_MODEK*) allow a user to set the mode of the tty driver. Functions *TC_ECHO* and *TC_NOECHO* control character echo, allowing a caller to turn off echo, accept input, and then turn echo back on. Finally, function *TC_ICHARS* allows a user to query the driver to determine how many characters are waiting in the input queue.

Observant readers may have noticed that neither parameter *arg1* nor *arg2* is used in function *ttyControl*. They have been declared, however, because the device-independent I/O routine *control* always provides four arguments when calling *ttyControl*. Although the compiler cannot perform type-checking on indirect calls, omitting the argument declarations makes the code less portable and more difficult to understand.

## 15.19 Perspective

The length of the code in the chapter reveals an important point about device drivers. To understand the point, compare the amount of code used for a trivial serial device to the code used for message passing and process synchronization primitives (i.e., semaphores). Although message passing and semaphores each provide a powerful abstraction, the code is relatively small.

Why does a trivial device driver contain so much code? After all, the driver only makes it possible to read and write characters. The answer lies in the difference between the abstraction the hardware supplies and the abstraction the driver provides. The underlying hardware merely transfers characters, and the output side is independent of the input side. Thus, the hardware does not handle flow control or character echo. Furthermore, the hardware knows nothing about end-of-line translation (i.e., the *crlf* mapping). Consequently, a driver must contain code that handles many details.

Although it may seem complex, the example driver in this chapter is trivial. A production device driver may comprise more than ten thousand lines of source code, and may contain hundreds of functions. Drivers for devices that can be plugged in at

run time (e.g., a USB device) are even more complex than drivers for static devices. Thus, one should appreciate that taken as a whole, code for device drivers is both huge and extremely complex.

## 15.20 Summary

A device driver consists of a set of functions that control a peripheral hardware device. The driver routines are partitioned into two halves: an upper half that contains the functions called from applications, and a lower half that contains functions that the system calls when a device interrupts. The two halves communicate through a shared data structure called a device control block.

The example device driver examined in this chapter is referred to as a tty driver. It manages input and output over serial line hardware, such as the connection to a keyboard. Upper-half functions in the example driver implement *read*, *write*, *getc*, *putc*, and *control* operations. Each upper-half function is called indirectly through the device-switch table. Lower-half functions in the example driver handle interrupts. During an output interrupt, the lower half fills the onboard FIFO from the echo or output queues. During an input interrupt, the lower half extracts and processes characters from the input FIFO.

## EXERCISES

**15.1** Predict what would happen if two processes executed *ttyRead* concurrently when both requested a large number of characters. Experiment and see what happens.

**15.2** *Kprintf* uses polled I/O: it disables interrupts, waits until the device is idle, displays its message, and then restores interrupts. What happens if the output buffer is full and *kprintf* is called repeatedly to display a *NEWLINE* character? Explain.

**15.3** Some systems partition asynchronous device drivers into three levels: interrupt level to do nothing but transfer characters to and from the device, upper level to transfer characters to and from the user, and a middle level to implement a *line discipline* that handles details like character echo, flow control, special processing, and out of band signals. Convert the Xinu tty driver to a three-level scheme, and arrange for processes to execute code in the middle layer.

**15.4** Suppose two processes both attempt to use *write()* on the CONSOLE device concurrently. What will the output be? Why?

**15.5** Implement a control function that allows a process to obtain exclusive use of the CONSOLE device and another control function that the process can use to release its use.

**15.6** *TtyControl* handles changes of mode poorly because it does not reset the cursor or buffer pointers. Rewrite the code to improve it. What happens to partially entered lines when changing from cooked to raw mode?

**15.7** When connecting two computers, it is useful to have flow control in both directions. Modify the tty driver to include a "tandem" mode that sends Control-S when the input buffer is nearly full, and then sends Control-Q when the buffer is half-empty.

**15.8** When a user changes the mode of a tty device, what should happen to characters already in the input queue (which were accepted before the mode changed)? One possibility is that the queue is discarded. Modify the code to implement discard during a mode change.

# Chapter Contents

# 16

# DMA Devices And Drivers (Ethernet)

*Modern hardware is stunningly difficult to deal with.*

— James Buchanan

## 16.1 Introduction

Previous chapters consider a general paradigm for I/O, and explain how a device driver is organized. Chapter 15 shows an example tty driver that illustrates how an upper half and a lower half interact.

This chapter extends our discussion of I/O by considering the design of device drivers for hardware devices that can transfer data from or to memory. The chapter uses an Ethernet interface as an example to show how the CPU informs the device about available buffers, and how the device accesses the buffers without requiring the processor to transfer data over the bus.

## 16.2 Direct Memory Access And Buffers

Although a bus can only transfer a word of data at one time, a block-oriented device, such as a disk or a network interface, needs to transfer multiple words of data to fill a given request. The motivation for *Direct Memory Access* (*DMA*) is parallelism: adding intelligence to an I/O device allows the device to perform multiple bus transfers without interrupting the CPU. Thus, a disk with DMA capability can transfer an entire disk block between memory and the device before interrupting the CPU, and a network interface can transfer an entire packet before interrupting the CPU.

DMA output is the easiest to understand. As an example, consider DMA output to a disk device. To write a disk block, the operating system places the data to be written in a buffer in memory, passes the buffer address to the disk device with a *write* request, and allows the CPU to continue executing. While the CPU executes, the disk DMA hardware uses the bus to transfer successive words of data from the buffer to the disk. Once an entire block has been read from memory and written to disk, the DMA hardware interrupts the CPU. If an additional disk block is ready for output, the operating system can start another DMA operation to transfer the block.

DMA input works the other way around. To read a disk block, the operating system allocates a buffer in memory and passes the buffer address to the device along with a request to *read* the block. After starting the DMA transfer, the operating system continues executing. Simultaneous with CPU execution, the DMA hardware uses the bus to transfer the block from the disk to the buffer in memory. Once the entire block has been written into the memory buffer, the DMA hardware interrupts the CPU. Thus, with DMA, only one interrupt occurs per block transferred.

## 16.3 Multiple Buffers And Rings

DMA devices are more complex than described above. Instead of passing the device the address of a buffer, the operating system allocates multiple buffers, links them together on a linked list, and passes the device the address of the list. The device hardware is designed to follow the linked list without waiting for the CPU to restart an operation. For example, consider a network interface that uses DMA hardware for input. To receive packets from the network, the operating system allocates a linked list of buffers that can each hold a network packet, and passes the address of the list to the network interface device. When a packet arrives, the network device moves to the next buffer on the list, uses DMA to copy the packet into the buffer, and then generates an interrupt. As long as buffers remain on the list, the device continues to accept incoming packets and place each packet in a buffer.

What happens if a DMA device reaches the end of a buffer list? Interestingly, most DMA devices never reach the end of the linked list because the hardware uses a circular linked list, called a *buffer ring*. That is, the last node on the list points back to the first. Each node in the list contains two values: a pointer to a buffer and a status bit that tells whether the buffer is ready for use. On input, the operating system initializes each node on the list to point to a buffer and sets the status to indicate *EMPTY*. When it fills a buffer, the DMA hardware changes the status to *FULL* and generates an interrupt. The device driver function that handles interrupts extracts the data from all buffers that are full, and clears the status bits to indicate that each buffer is *EMPTY*. On the one hand, if the operating system is fast enough, it will be able to process each incoming packet and mark the buffer *EMPTY* before another packet arrives. Thus, the DMA hardware will keep moving around the ring without ever encountering a buffer that is marked *FULL*. On the other hand, if the operating system cannot process packets as fast as they arrive, the device will eventually fill all the buffers and will encounter a buffer marked *FULL*. If it travels completely around the ring and encounters a full

buffer, the DMA hardware sets an error indicator (typically an *overflow* bit) and generates an interrupt to inform the operating system.

Most DMA hardware also uses a circular linked list of buffers for output. The operating system creates the ring with each buffer marked *EMPTY*. When it has a packet to send, the operating system places the packet in the next available output buffer, marks the buffer *FULL*, and starts the device if the device is not currently running. The device moves to the next buffer, extracts the packet, and transmits the packet. Once started, the DMA hardware continues to move around the ring until it reaches an empty buffer. Thus, if applications generate data fast enough, the DMA hardware will transmit packets continuously without ever encountering an empty buffer.

## 16.4 An Example Ethernet Driver Using DMA

An example will clarify the discussion above. Our example driver is written for an *Atheros AG71xx* Ethernet interface,† the Ethernet interface used in the Linksys E2100L. Although many of the details are specific to the Atheros device, the interaction between the processors and device is typical of most DMA devices.

The AG71xx performs both input and output, and the chip uses a separate DMA engine for each. That is, a driver must create two rings — one ring points to buffers used to receive packets, and the other points to buffers used to send packets. The device has separate registers that a driver uses to pass pointers to the rings, and the hardware allows input and output to proceed simultaneously. Despite operating independently, both input and output interrupts use a single interrupt vector. Therefore, when an interrupt occurs, device driver software must interact with the device to determine whether the interrupt corresponds to an input or output operation.

## 16.5 Device Hardware Definitions And Constants

File *ag71xx.h* defines constants and structures for the AG71xx hardware. The file contains many details and may seem confusing. For now, it is sufficient to know that the definitions are taken directly from the vendor's manual for the device.

†The characters *xx* in the name indicate that the manufacturer offers a series of products that all have the same API.

```
/* ag71xx.h  -  Definitions for an Atheros ag71xx Ethernet device       */

/* Ring buffer sizes */

#define ETH_RX_RING_ENTRIES 64  /* Number of buffers on Rx Ring */
#define ETH_TX_RING_ENTRIES 128 /* Number of buffers on Tx Ring */

#define ETH_PKT_RESERVE     64

/* Control and Status register layout for the ag71xx  */

struct ag71xx {
    volatile uint32 macConfig1;         /* 0x000 MAC configuration 1    */

#define MAC_CFG1_TX        (1 <<  0)    /* Enable Transmitter           */
#define MAC_CFG1_SYNC_TX   (1 <<  1)    /* Syncronize Transmitter       */
#define MAC_CFG1_RX        (1 <<  2)    /* Enable Receiver              */
#define MAC_CFG1_SYNC_RX   (1 <<  3)    /* Syncronize Receiver          */
#define MAC_CFG1_LOOPBACK  (1 <<  8)    /* Enable Loopback              */
#define MAC_CFG1_SOFTRESET (1 << 31)    /* Software Reset               */

    volatile uint32 macConfig2;         /* 0x004 MAC configuration 2    */
#define MAC_CFG2_FDX       (1 <<  0)    /* Enable Full Duplex           */
#define MAC_CFG2_CRC       (1 <<  1)    /* Enable CRC appending         */
#define MAC_CFG2_PAD       (1 <<  2)    /* Enable padding of short pkts */
#define MAC_CFG2_LEN_CHECK (1 <<  4)    /* Enable length field checking */
#define MAC_CFG2_HUGE      (1 <<  5)    /* Enable frames longer than max*/
#define MAC_CFG2_IMNIBBLE  (1 <<  8)    /* "nibble mode" interface type */
#define MAC_CFG2_IMBYTE    (2 <<  8)    /* "byte mode" interface type   */


    volatile uint32 pad00[2];
    volatile uint32 pad01[4];
    volatile uint32 pad02[4];
    volatile uint32 pad03[4];

    volatile uint32 macAddr1;           /* 0x040 MAC Address part 1     */
    volatile uint32 macAddr2;           /* 0x044 MAC Address part 2     */

    volatile uint32 fifoConfig0;        /* 0x048 MAC configuration 0    */

#define FIFO_CFG0_WTMENREQ (1 <<  8)    /* Enable FIFO watermark module */
#define FIFO_CFG0_SRFENREQ (1 <<  9)    /* Enable FIFO system Rx module */
#define FIFO_CFG0_FRFENREQ (1 << 10)    /* Enable FIFO fabric Rx module */
#define FIFO_CFG0_STFENREQ (1 << 11)    /* Enable FIFO system Tx module */
```

```
#define FIFO_CFG0_FTFENREQ (1 << 12)    /* Enable FIFO fabric Tx module */

    volatile uint32 fifoConfig1;        /* 0x04C MAC configuration 1    */
    volatile uint32 fifoConfig2;        /* 0x050 MAC configuration 2    */
    volatile uint32 fifoConfig3;        /* 0x054 MAC configuration 3    */
    volatile uint32 fifoConfig4;        /* 0x058 MAC configuration 4    */
    volatile uint32 fifoConfig5;        /* 0x05C MAC configuration 5    */

    volatile uint32 pad06[72];

    volatile uint32 txControl;          /* 0x180 Tx Control             */

#define TX_CTRL_ENABLE     (1 <<  0)    /* Enable Tx                    */
    volatile uint32 txDMA;              /* 0x184 Tx DMA Descriptor      */
    volatile uint32 txStatus;           /* 0x188 Tx Status              */

#define TX_STAT_SENT       (1 <<  0)    /* Packet Sent                  */
#define TX_STAT_UNDER      (1 <<  1)    /* Tx Underrun                  */

    volatile uint32 rxControl;          /* 0x18C Rx Control             */

#define RX_CTRL_RXE        (1 <<  0)    /* Enable receiver              */

    volatile uint32 rxDMA;              /* 0x190 Rx DMA Descriptor      */
    volatile uint32 rxStatus;           /* 0x194 Rx Status              */

#define RX_STAT_RECVD      (1 <<  0)    /* Packet Received              */
#define RX_STAT_OVERFLOW   (1 <<  2)    /* DMA Rx overflow              */
#define RX_STAT_COUNT   (0xFF << 16)    /* Count of packets received    */

    volatile uint32 interruptMask;      /* 0x198 Interrupt Mask         */

#define IRQ_TX_PKTSENT     (1 <<  0)    /* Packet Sent                  */
#define IRQ_TX_UNDERFLOW   (1 <<  1)    /* Tx packet underflow          */
#define IRQ_TX_BUSERR      (1 <<  3)    /* Tx Bus Error                 */
#define IRQ_RX_PKTRECV     (1 <<  4)    /* Rx Packet received           */
#define IRQ_RX_OVERFLOW    (1 <<  6)    /* Rx Overflow                  */
#define IRQ_RX_BUSERR      (1 <<  7)    /* Rx Bus Error                 */

    volatile uint32 interruptStatus;    /* 0x19C Interrupt Status       */
};

/* Receiver header struct and constants */

#define ETH_RX_FLAG_OFIFO   0x0001      /* FIFO Overflow                */
```

```
#define ETH_RX_FLAG_CRCERR  0x0002      /* CRC Error                      */
#define ETH_RX_FLAG_SERR    0x0004      /* Receive Symbol Error           */
#define ETH_RX_FLAG_ODD     0x0008      /* Frame has odd number nibbles */
#define ETH_RX_FLAG_LARGE   0x0010      /* Frame is > RX MAX Length       */
#define ETH_RX_FLAG_MCAST   0x0020      /* Dest is Multicast Address      */
#define ETH_RX_FLAG_BCAST   0x0040      /* Dest is Broadcast Address      */
#define ETH_RX_FLAG_MISS    0x0080      /* Received due to promisc mode */
#define ETH_RX_FLAG_LAST    0x0800      /* Last buffer in frame           */
#define ETH_RX_FLAG_ERRORS  ( ETH_RX_FLAG_ODD | ETH_RX_FLAG_SERR | \
                              ETH_RX_FLAG_CRCERR | ETH_RX_FLAG_OFIFO )

/* Header on a received packet */

struct rxHeader {
    uint16 length;                      /* Length of packet data          */
    uint16 flags;                       /* Receive flags                  */
    uint16 pad[12];                     /* Padding                        */
};

/* Ethernet DMA descriptor */

#define ETH_DESC_CTRL_LEN   0x00001fff  /* Mask for length field          */
#define ETH_DESC_CTRL_MORE  0x10000000  /* More fragments                 */
#define ETH_DESC_CTRL_EMPTY 0x80000000  /* Empty descriptor               */

/* Descriptor for the DMA engine to determine where to  */
/*      find a packet buffer.                           */

struct dmaDescriptor {
    uint32 address;                     /* Stored as physical address     */
    uint32 control;                     /* DMA control bits               */
    uint32 next;                        /* Next descriptor in the ring    */
};

#define RESET_CORE 0xB806001C           /* Atheros bus core reset reg     */
#define RESET_E0_MAC (1 <<  9)          /* Reset Ethernet zero MAC        */
#define RESET_E1_MAC (1 << 13)          /* Reset Ethernet one  MAC        */
```

Struct *ag71xx* specifies the format of control and status registers for the AG71xx hardware. Many of the fields are labeled *volatile* to inform the compiler that each reference to the field must result in a fetch operation (i.e., the compiler cannot apply an optimization that fetches the item once, stores the value in a register, and reuses the value from the register).

## 16.6 Rings And Buffers In Memory

From a device's perspective, an input or output ring consists of a linked list in memory. Each node on the list is defined using struct *dmaDescriptor*, which contains three items: a pointer to a buffer in memory, a status word used to tell whether the buffer currently contains data, and a pointer to the next node on the list. Figure 16.1 illustrates how the transmit and receive rings are organized and how each node in the ring contains a pointer to a buffer.

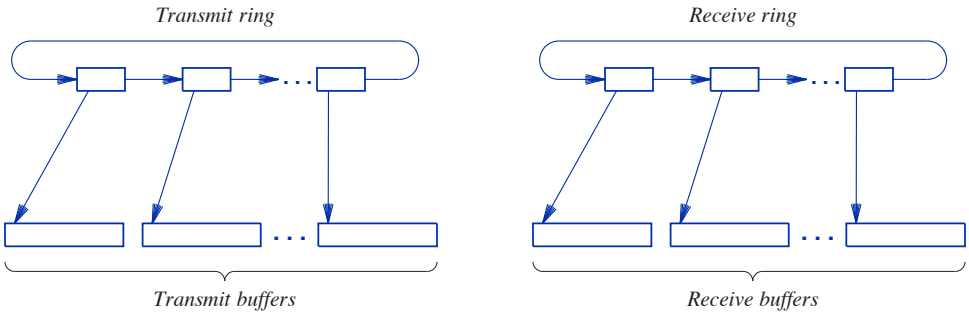



**Figure 16.1** Illustration of the transmit and receive rings used with the example DMA hardware device.

As the figure illustrates, each ring consists of a circular linked list with a pointer from a node to its successor (and the final node pointing back to the first). When reading the code, it will be important to remember that the Ethernet device views the rings as a linked list (i.e., the device follows the pointer in a node to get to the next node on the list). The reason the distinction is important arises from the way our driver code allocates storage. The driver places ring nodes in contiguous storage (i.e., nodes of the list are allocated in an array). Therefore, the driver can use array indexing to move through nodes. Figure 16.2 shows the structure of a node and illustrates how nodes of a ring are stored in an array.

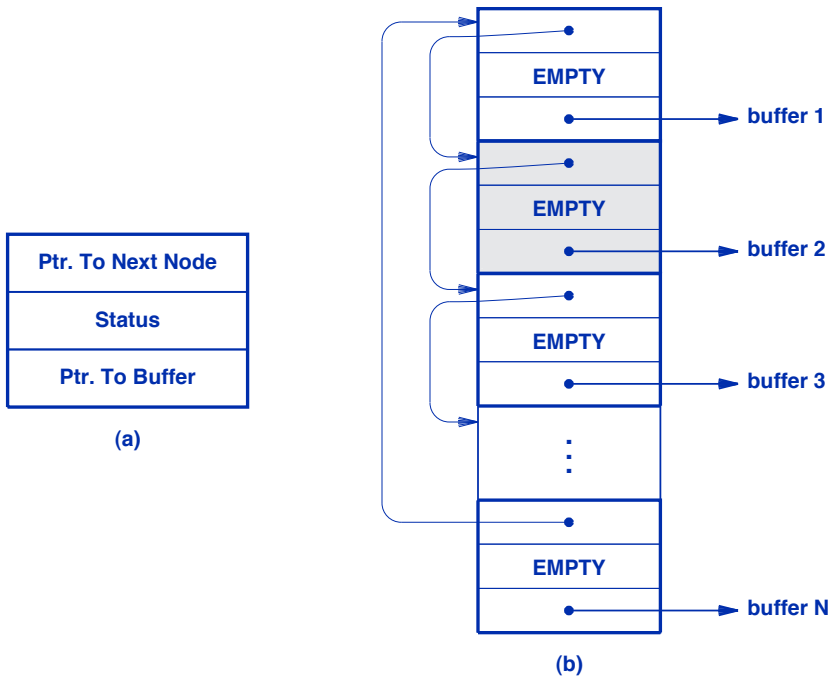



**Figure 16.2** (a) The contents of a node on a receive or transmit ring, and (b) a ring stored in an array with the second node shaded.

## 16.7 Definitions Of An Ethernet Control Block

File *ether.h* defines constants and data structures used by the Ethernet driver, including the format of an Ethernet packet header, the layout of a packet buffer in memory, and the contents of an Ethernet control block.

```
/* ether.h */

/* Ethernet packet format:

 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Dest. MAC (6)  |  Src. MAC (6)   |Type (2)|      Data (46-1500)...    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
*/

#define ETH_ADDR_LEN    6       /* Length of Ethernet (MAC) address     */

/* Ethernet packet header */
```

```
struct  etherPkt {
        byte    dst[ETH_ADDR_LEN];      /* Destination Mac address      */
        byte    src[ETH_ADDR_LEN];      /* Source Mac address           */
        uint16  type;                   /* Ether type field             */
        byte    data[1];                /* Packet payload               */
};

#define ETH_HDR_LEN     14      /* Length of Ethernet packet header     */

/* Ethernet Buffer lengths */

#define ETH_IBUFSIZ     1024    /* input buffer size                    */

/* Ethernet DMA buffer sizes */

#define ETH_MTU         1500    /* Maximum transmission unit            */
#define ETH_VLAN_LEN    4       /* Length of Ethernet vlan tag          */
#define ETH_CRC_LEN     4       /* Length of CRC on Ethernet frame      */

#define ETH_MAX_PKT_LEN ( ETH_HDR_LEN + ETH_VLAN_LEN + ETH_MTU )

#define ETH_RX_BUF_SIZE ( ETH_MAX_PKT_LEN + ETH_CRC_LEN \
                                + sizeof(struct rxHeader) )

#define ETH_TX_BUF_SIZE ( ETH_MAX_PKT_LEN )

/* State of the Ethernet interface */

#define ETH_STATE_FREE  0       /* control block is unused */
#define ETH_STATE_DOWN  1       /* interface is currently inactive */
#define ETH_STATE_UP    2       /* interface is currently active */

/* Ethernet device control functions */

#define ETH_CTRL_CLEAR_STATS 1  /* Reset Ethernet Statistics            */
#define ETH_CTRL_SET_MAC     2  /* Set the MAC for this device          */
#define ETH_CTRL_GET_MAC     3  /* Get the MAC for this device          */
#define ETH_CTRL_SET_LOOPBK  4  /* Set Loopback Mode                    */
#define ETH_CTRL_RESET       5  /* Reset the Ethernet device            */
#define ETH_CTRL_DISABLE     6  /* Disable the Ethernet device          */

/* Ethernet packet buffer */

struct  ethPktBuffer {
```

```
        byte    *buf;           /* Pointer to a packet buffer           */
        byte    *data;          /* Start of data within the buffer      */
        int32   length;         /* Length of data in the packet buffer  */
};

/* Ethernet control block */

#define ETH_INVALID     (-1)    /* Invalid data (virtual devices)       */

struct  ether   {
        byte    state;          /* ETH_STATE_... as defined above       */
        struct  dentry  *phy;   /* physical ethernet device for Tx DMA  */

        /* Pointers to associated structures */

        struct  dentry  *dev;   /* address in device switch table       */
        void    *csr;           /* addr.of control and status regs.     */

        uint32  interruptMask;  /* interrupt mask                       */
        uint32  interruptStatus;/* interrupt status                     */

        struct  dmaDescriptor *rxRing;/* array of receive ring descrip. */
        struct  ethPktBuffer **rxBufs;/* Rx ring array                  */
        uint32  rxHead;         /* Index of current head of Rx ring     */
        uint32  rxTail;         /* Index of current tail of Rx ring     */
        uint32  rxRingSize;     /* size of Rx ring descriptor array     */
        uint32  rxirq;          /* Count of Rx interrupt requests       */
        uint32  rxOffset;       /* Size in bytes of rxHeader            */
        uint32  rxErrors;       /* Count of Rx errors                   */

        struct  dmaDescriptor *txRing;/* array of transmit ring descrip.*/
        struct  ethPktBuffer **txBufs;/* Tx ring array                  */
        uint32  txHead;         /* Index of current head of Tx ring     */
        uint32  txTail;         /* Index of current tail of Tx ring     */
        uint32  txRingSize;     /* size of Tx ring descriptor array     */
        uint32  txirq;          /* Count of Tx interrupt requests       */

        byte    devAddress[ETH_ADDR_LEN]; /* MAC address */

        byte    addressLength;  /* Hardware address length              */
        uint16  mtu;            /* Maximum transmission unit (payload)  */

        uint32  errors;         /* Number of Ethernet errors            */
        uint16  ovrrun;         /* Buffer overruns                      */
        sid32   isema;          /* I/O semaphore for Ethernet input     */
```

```
        uint16  istart;         /* Index of packet in the input buffer  */
        uint16  icount;         /* Count of packets in the input buffer */

        struct  ethPktBuffer *in[ETH_IBUFSIZ]; /* Input buffer          */

        int     inPool;         /* Buffer pool ID for input buffers     */
        int     outPool;        /* Buffer pool ID for output buffers    */
};
extern  struct  ether   ethertab[];     /* array of control blocks */

int32   colon2mac(char *, byte *);
int32   allocRxBuffer(struct ether *, int32);
int32   waitOnBit(volatile uint32 *, uint32, const int32, int32);
```

## 16.8 Device And Driver Initialization

At system startup, the operating system calls function *ethInit* to initialize the Ethernet device and the device driver data structures. File *ethInit.c* contains the code.

```
/* ethInit.c - ethInit */

#include <xinu.h>

struct  ether   ethertab[Neth];         /* Ethernet control blocks */

/*------------------------------------------------------------------------
 * udelay - microsecond delay loop (CPU loop)
 *------------------------------------------------------------------------
 */
void    udelay(uint32 n) {

        uint32  delay;                  /* amount to delay measured in  */
                                        /*   clock cycles               */
        uint32  start = 0;              /* clock at start of delay      */
        uint32  target = 0;             /* computed clk at end of delay */
        uint32  count = 0;              /* current clock during loop    */

        delay = 200 * n;                /* 200 CPU cycles per usec */

        start = clkcount();             /* Get current clock */
        target = start + delay;         /* Compute finish time */

        if (target >= start) {
                while (((count = clkcount()) < target) &&
                                (count >= start)) {
                        ;  /* spin doing nothing */
                }
        } else {

                /* need to wrap around counter */

                while ((count = clkcount()) > start) {
                        ;  /* spin doing nothing */
                }
                while ((count = clkcount()) < target) {
                        ;  /* spin doing nothing */
                }
        }
}

/*------------------------------------------------------------------------
 * mdelay - millisecond delay loop (CPU loop)
 *------------------------------------------------------------------------
 */
```

```
void    mdelay(uint32 n) {
        int i;

        for (i = 0; i < n; i++) {
                udelay(1000);
        }
}

/*-------------------------------------------------------------------------
 * ethInit - Initialize Ethernet device structures
 *-------------------------------------------------------------------------
 */
devcall ethInit (
         struct dentry *devptr
        )
{
        struct  ether   *ethptr;
        struct  ag71xx  *nicptr;
        uint32  *rstptr;
        uint32  rstbit;

        /* Initialize structure pointers */

        ethptr = &ethertab[devptr->dvminor];
        memset(ethptr, '\0', sizeof(struct ether));
        ethptr->dev = devptr;
        ethptr->csr = devptr->dvcsr;

        /* Get device CSR address */

        nicptr = (struct ag71xx *)devptr->dvcsr;
        rstptr = (uint32 *)RESET_CORE;
        if (devptr->dvminor == 0) {     /* use E0 on first device only */
                rstbit = RESET_E0_MAC;
        } else {
                rstbit = RESET_E1_MAC;
        }

        ethptr->state = ETH_STATE_DOWN;
        ethptr->rxRingSize = ETH_RX_RING_ENTRIES;
        ethptr->txRingSize = ETH_TX_RING_ENTRIES;
        ethptr->mtu = ETH_MTU;
        ethptr->interruptMask = IRQ_TX_PKTSENT | IRQ_TX_BUSERR
                | IRQ_RX_PKTRECV | IRQ_RX_OVERFLOW | IRQ_RX_BUSERR;
```

```
ethptr->errors = 0;
ethptr->isema = semcreate(0);
ethptr->istart = 0;
ethptr->icount = 0;
ethptr->ovrrun = 0;
ethptr->rxOffset = ETH_PKT_RESERVE;

colon2mac(nvramGet("et0macaddr"), ethptr->devAddress);
ethptr->addressLength = ETH_ADDR_LEN;

/* Reset the device */

nicptr->macConfig1 |= MAC_CFG1_SOFTRESET;
udelay(20);
*rstptr |= rstbit;
mdelay(100);
*rstptr &= ~rstbit;
mdelay(100);

/* Enable transmit and receive */

nicptr->macConfig1 = MAC_CFG1_TX | MAC_CFG1_SYNC_TX |
        MAC_CFG1_RX | MAC_CFG1_SYNC_RX;

/* Configure full duplex, auto padding CRC, */
/*      and interface mode                  */

nicptr->macConfig2 |= MAC_CFG2_FDX | MAC_CFG2_PAD |
                MAC_CFG2_LEN_CHECK | MAC_CFG2_IMNIBBLE;

/* Enable FIFO modules */

nicptr->fifoConfig0 = FIFO_CFG0_WTMENREQ | FIFO_CFG0_SRFENREQ |
  FIFO_CFG0_FRFENREQ | FIFO_CFG0_STFENREQ | FIFO_CFG0_FTFENREQ;

nicptr->fifoConfig1 = 0x0FFF0000;

/* Max out number of words to store in Receiver RAM */

nicptr->fifoConfig2 = 0x00001FFF;

/* Drop any incoming packet with errors in the Rx stats vector  */

nicptr->fifoConfig4 = 0x0003FFFF;
```

```
/* Drop short packets (set "don't care" on Rx stats vector bits */

nicptr->fifoConfig5 = 0x0003FFFF;

/* Buffers should be page-aligned and cache-aligned */

ethptr->rxBufs = (struct ethPktBuffer **)getstk(PAGE_SIZE);
ethptr->txBufs = (struct ethPktBuffer **)getstk(PAGE_SIZE);
ethptr->rxRing = (struct dmaDescriptor *)getstk(PAGE_SIZE);
ethptr->txRing = (struct dmaDescriptor *)getstk(PAGE_SIZE);

if (  ( (int32)ethptr->rxBufs == SYSERR )
   || ( (int32)ethptr->txBufs == SYSERR )
   || ( (int32)ethptr->rxRing == SYSERR )
   || ( (int32)ethptr->txRing == SYSERR ) ) {
        return SYSERR;
}

/* Translate buffer and ring pointers to KSEG1 */

ethptr->rxBufs = (struct ethPktBuffer **)
        (((uint32)ethptr->rxBufs - PAGE_SIZE +
          sizeof(int32)) | KSEG1_BASE);
ethptr->txBufs = (struct ethPktBuffer **)
        (((uint32)ethptr->txBufs - PAGE_SIZE +
          sizeof(int32)) | KSEG1_BASE);
ethptr->rxRing = (struct dmaDescriptor *)
        (((uint32)ethptr->rxRing - PAGE_SIZE +
          sizeof(int32)) | KSEG1_BASE);
ethptr->txRing = (struct dmaDescriptor *)
        (((uint32)ethptr->txRing - PAGE_SIZE +
          sizeof(int32)) | KSEG1_BASE);

/* Set buffer pointers and rings to zero */

memset(ethptr->rxBufs, '\0', PAGE_SIZE);
memset(ethptr->txBufs, '\0', PAGE_SIZE);
memset(ethptr->rxRing, '\0', PAGE_SIZE);
memset(ethptr->txRing, '\0', PAGE_SIZE);

/* Initialize the interrupt vector and enable the device */

interruptVector[devptr->dvirq] = devptr->dvintr;
enable_irq(devptr->dvirq);
```

```
        return OK;
}
```

The hardware requires specific delays between certain initialization steps (to give the hardware on the device sufficient time to perform the step). Because the operating system is not running when the initialization code executes, delay functions such as *sleep* are not available. Therefore, the code defines two delay functions, *udelay* and *mdelay*, that delay for a specified number of microseconds and milliseconds. In each case, the code consists of a loop that keeps the CPU occupied for the specified time. Both functions are platform dependent because the number of times a loop must iterate depends on the clock speed of the CPU. More important, the rate clock on a given copy of the hardware may differ slightly from the clock rate on another. To accommodate differences, our implementation uses the real-time clock to adjust the delay. That is, a delay function reads the clock, estimates the number of times a loop should execute, and runs the loop. It then re-reads the clock to determine whether additional delay is needed.

When it is called, *ethInit* initializes fields in the device control block, and then initializes the hardware. Many of the details depend on the specific Ethernet hardware, but one item applies broadly to most DMA hardware devices: the addressing mode. As on most systems, a DMA hardware device on the E2100L uses the underlying bus directly. Thus, the hardware device will use physical addresses rather than the segment addresses used by the operating system. Consequently, when passing an address to the device, the operating system must convert all addresses to physical addresses. In particular, physical addresses must be stored in the buffer rings and must be used when passing the address of a buffer ring to the device. In the code, translation from a segment address to a physical address is achieved by an inline function that computes the *logical or* of an address with the constant *KSEG1_BASE*. Although the details of translation vary among platforms (and may require the operating system to use MMU hardware), the concept remains the same: when linked lists are used with DMA, each address must be translated to a form the hardware can understand.

As a final step, *ethInit* enables device interrupts (i.e., allows the device to begin to receive and store packets and generate interrupts). Enabling device interrupts does not mean that interrupts can occur: during initialization, the operating system runs a mode where all CPU interrupts are disabled. Thus, the device can request an interrupt, but the CPU will not process the interrupt. Once the operating system enables CPU interrupts, the device will be able to interrupt the CPU and deliver packets that have accumulated.

## 16.9 Allocating An Input Buffer

Before a packet can be read, the driver must allocate a buffer to hold the packet and link the buffer into the ring used for DMA input. Our driver uses a utility function, *allocRxBuffer* to allocate a buffer from the buffer pool and link the buffer into the DMA ring. File *allocRxBuffer.c* contains the code.

```
/* allocRxBuffer.c - allocRxBuffer */

#include <xinu.h>

/*------------------------------------------------------------------------
 * allocRxBuffer - allocate an Ethernet packet buffer structure
 *------------------------------------------------------------------------
 */
int32   allocRxBuffer (
         struct ether   *ethptr,        /* ptr to device control block  */
         int32  destIndex               /* index in receive ring        */
        )
{
        struct  ethPktBuffer    *pkt;
        struct  dmaDescriptor   *dmaptr;

        /* Compute next ring location modulo the ring size */

        destIndex %= ethptr->rxRingSize;

        /* Allocate a packet buffer */

        pkt = (struct ethPktBuffer *)getbuf(ethptr->inPool);

        if ((uint32)pkt == SYSERR) {
                kprintf("eth0 allocRxBuffer() error\r\n");
                return SYSERR;
        }
        pkt->length = ETH_RX_BUF_SIZE;
        pkt->buf = (byte *)(pkt + 1);

        /* Data region offset by size of rx header */

        pkt->data = pkt->buf + ethptr->rxOffset;

        ethptr->rxBufs[destIndex] = pkt;

        /* Fill in DMA descriptor fields */

        dmaptr = ethptr->rxRing + destIndex;
        dmaptr->control = ETH_DESC_CTRL_EMPTY;
        dmaptr->address = (uint32)(pkt->buf) & PMEM_MASK;

        return OK;
}
```

In our conceptual description of DMA, we said that each node in a ring contains a bit that tells whether the buffer is full or empty. The AG71xx hardware uses the *control* field in the DMA descriptor (struct *dmaDescriptor*) to contain the bit. Symbolic constant *ETH_DESC_CTRL_EMPTY* specifies that the bit is 1 when the buffer is empty and 0 when the buffer contains a packet.

Thus, when it allocates a buffer and links the buffer into a ring slot, *allocRxBuffer* must set bit *ETH_DESC_CTRL_EMPTY* to indicate that the buffer is empty; the device hardware will set the bit to 0 when a packet has arrived and been placed in the buffer. *AllocRxBuffer* takes two arguments: a pointer to the Ethernet control block and the index of the ring slot to use. After it allocates a buffer, *allocRxBuffer* computes the address of the ring slot, places the buffer address in the header, and sets the *control* field.

## 16.10 Reading From An Ethernet Device

Because the DMA engine uses the input ring to store incoming packets in successive buffers, reading from an Ethernet device does not involve much interaction with the device hardware. Instead, the driver uses a semaphore to coordinate reading: an application process waits on the semaphore, and the interrupt code signals the semaphore when a packet arrives. Thus, if no packet is available, a caller will block on the semaphore. Once a packet is available, the interrupt handler signals the semaphore, and the caller proceeds. The packet will have been placed in the next ring buffer. The driver function that handles reading merely needs to copy the packet from the ring buffer to the caller's buffer and return. File *ethRead.c* contains the code:

```
/* ethRead.c  - ethRead */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethRead - read a packet from an Ethernet device
 *------------------------------------------------------------------------
 */
devcall ethRead (
         struct dentry  *devptr,        /* entry in device switch table */
         void   *buf,                   /* buffer to hold packet        */
         uint32 len                     /* length of buffer             */
        )
{
        struct  ether   *ethptr;        /* ptr to entry in ethertab     */
        struct  ethPktBuffer    *pkt;   /* ptr to a packet              */
        uint32  length;                 /* packet length                */

        ethptr = &ethertab[devptr->dvminor];
```

```
        if (ETH_STATE_UP != ethptr->state) {
                return SYSERR; /* interface is down */
        }

        /* Make sure user's buffer is large enough to store at least    */
        /*   the header of a packet                                     */

        if (len < ETH_HDR_LEN) {
                return SYSERR;
        }

        /* Wait for a packet to arrive */

        wait(ethptr->isema);

        /* Pick up packet */

        pkt = ethptr->in[ethptr->istart];
        ethptr->in[ethptr->istart] = NULL;
        ethptr->istart = (ethptr->istart + 1) % ETH_IBUFSIZ;
        ethptr->icount--;

        if (pkt == NULL) {
                return 0;
        }

        length = pkt->length;
        memcpy(buf, (byte *)(((uint32)pkt->buf) | KSEG1_BASE), length);
        freebuf((char *)pkt);

        return length;
}
```

After verifying that the Ethernet device is up and checking its arguments, *ethRead* waits on the input semaphore. The call to *wait* blocks until at least one packet is available. Once the function proceeds beyond the call to *wait*, the function only needs to locate the next available ring buffer, copy the packet to the caller's buffer, and return. Field *istart* in the device driver control block gives the index of the ring buffer to use. (Remember that even though the device hardware views the buffer ring as a linked list, the driver uses array indexing.) Once it uses the entry, *ethRead* moves *istart* to the next ring buffer by adding 1 modulo the number of ring buffers.

## 16.11 Writing To An Ethernet Device

Using DMA makes output as straightforward as input. An application calls *write* to send a packet, which invokes function *ethWrite*. As with input, the output side only interacts with the ring buffers: *ethWrite* copies the caller's buffer to the next available output buffer. File *ethWrite.c* contains the code:

```
/* ethWrite.c -  etherWrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethWrite - write a packet to an Ethernet device
 *------------------------------------------------------------------------
 */
devcall ethWrite (
         struct dentry  *devptr,        /* entry in device switch table */
         void   *buf,                   /* buffer to hold packet        */
         uint32 len                     /* length of buffer             */
        )
{
        struct  ether   *ethptr;
        struct  ag71xx  *nicptr;
        struct  ethPktBuffer *pkt;
        struct  dmaDescriptor *dmaptr;
        uint32  tail = 0;
        byte    *buffer;

        buffer = buf;

        ethptr = &ethertab[devptr->dvminor];
        nicptr = ethptr->csr;

        if ((ETH_STATE_UP != ethptr->state)
                || (len < ETH_HDR_LEN)
                || (len > (ETH_TX_BUF_SIZE - ETH_VLAN_LEN))) {
                return SYSERR;
        }

        tail = ethptr->txTail % ETH_TX_RING_ENTRIES;
        dmaptr = &ethptr->txRing[tail];

        if (!(dmaptr->control & ETH_DESC_CTRL_EMPTY)) {
                ethptr->errors++;
```

```
                return SYSERR;
        }

        pkt = (struct ethPktBuffer *)getbuf(ethptr->outPool);
        if ((uint32)pkt == SYSERR) {
                ethptr->errors++;
                return SYSERR;
        }

        /* Translate pkt pointer into uncached memory space */

        pkt = (struct ethPktBuffer *)((int)pkt | KSEG1_BASE);
        pkt->buf = (byte *)(pkt + 1);
        pkt->data = pkt->buf;
        memcpy(pkt->data, buffer, len);

        /* Place filled buffer in outgoing queue */
        ethptr->txBufs[tail] = pkt;

        /* Add the buffer to the transmit ring.  Note that the address  */
        /*  must be physical (USEG) because the DMA engine will used it */

        ethptr->txRing[tail].address = (uint32)pkt->data & PMEM_MASK;

        /* Clear empty flag and write the length */

        ethptr->txRing[tail].control = len & ETH_DESC_CTRL_LEN;

        /* move to next position */

        ethptr->txTail++;

        if (nicptr->txStatus & TX_STAT_UNDER) {
                nicptr->txDMA = ((uint32)(ethptr->txRing + tail))
                        & PMEM_MASK;
                nicptr->txStatus = TX_STAT_UNDER;
        }

        /* Enable transmit interrupts */

        nicptr->txControl = TX_CTRL_ENABLE;
        return len;
}
```

After checking its arguments, *ethWrite* waits for a free slot in the output ring and copies a packet from the caller's buffer into the ring buffer. If the device is currently idle, *ethWrite* must start the device. Starting a DMA device is trivial: *ethWrite* assigns constant *TX_CTRL_ENABLE* to the device's transmit control register. If the device is already running, the assignment has no effect; if the device is idle, the assignment starts the device at the next ring slot (i.e., the slot into which the driver placed the packet to be transmitted).

## 16.12 Handling Interrupts From An Ethernet Device

One of the advantages of a DMA device arises because the DMA engine on the device handles many of the details. As a result, interrupt processing does not involve much interaction with the device. An interrupt occurs when an input or output operation completes successfully or when the DMA engine encounters an error. The interrupt handler interrogates the device to determine the cause of an interrupt. For a successful input interrupt, the handler calls function *rxPackets*, and for a successful output interrupt, the handler calls function *txPackets*. Errors conditions are handled directly. File *ethInterrupt.c* contains the code:

```
/* ethInterrupt.c - ethInterrupt */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rxPackets - handler for receiver interrupts
 *------------------------------------------------------------------------
 */
void    rxPackets (
         struct ether *ethptr,          /* ptr to control block         */
         struct ag71xx *nicptr          /* ptr to device CSRs           */
        )
{
        struct  dmaDescriptor *dmaptr;  /* ptr to DMA descriptor        */
        struct  ethPktBuffer  *pkt;     /* ptr to one packet buffer     */
        int32   head;


        /* Move to next packet, wrapping around if needed */

        head = ethptr->rxHead % ETH_RX_RING_ENTRIES;
        dmaptr = &ethptr->rxRing[head];
        if (dmaptr->control & ETH_DESC_CTRL_EMPTY) {
                nicptr->rxStatus = RX_STAT_RECVD;
```

```
                return;
        }

        pkt = ethptr->rxBufs[head];
        pkt->length = dmaptr->control & ETH_DESC_CTRL_LEN;

        if (ethptr->icount < ETH_IBUFSIZ) {
                allocRxBuffer(ethptr, head);
                ethptr->in[(ethptr->istart + ethptr->icount) %
                                ETH_IBUFSIZ] =  pkt;
                ethptr->icount++;
                signal(ethptr->isema);
        } else {
                ethptr->ovrrun++;
                memset(pkt->buf, '\0', pkt->length);
        }

        ethptr->rxHead++;

        /* Clear the Rx interrupt */

        nicptr->rxStatus = RX_STAT_RECVD;
        return;
}

/*------------------------------------------------------------------------
 * txPackets - handler for transmitter interrupts
 *------------------------------------------------------------------------
 */
void    txPackets (
         struct ether *ethptr,          /* ptr to control block         */
         struct ag71xx *nicptr          /* ptr to device CSRs           */
        )
{
        struct  dmaDescriptor *dmaptr;
        struct  ethPktBuffer **epb = NULL;
        struct  ethPktBuffer *pkt = NULL;
        uint32  head;

        if (ethptr->txHead == ethptr->txTail) {
                nicptr->txStatus = TX_STAT_SENT;
                return;
        }

        /* While packets remain to be transmitted */
```

```
        while (ethptr->txHead != ethptr->txTail) {
                head = ethptr->txHead % ETH_TX_RING_ENTRIES;
                dmaptr = &ethptr->txRing[head];
                if (!(dmaptr->control & ETH_DESC_CTRL_EMPTY)) {
                        break;
                }

                epb = &ethptr->txBufs[head];

                /* Clear the Tx interrupt */

                nicptr->txStatus = TX_STAT_SENT;

                ethptr->txHead++;
                pkt = *epb;
                if (NULL == pkt) {
                        continue;
                }
                freebuf((void *)((bpid32)pkt & (PMEM_MASK | KSEG0_BASE)));
                *epb = NULL;
        }
        return;
}

/*------------------------------------------------------------------------
 * ethInterrupt - decode and handle interrupt from an Ethernet device
 *------------------------------------------------------------------------
 */
interrupt ethInterrupt(void)
{
        struct  ether   *ethptr;        /* ptr to control block         */
        struct  ag71xx  *nicptr;        /* ptr to device CSRs           */
        uint32  status;
        uint32  mask;

        /* Initialize structure pointers */

        ethptr = &ethertab[0];          /* default physical Ethernet    */
        if (!ethptr) {
                return;
        }
        nicptr = ethptr->csr;
        if (!nicptr) {
                return;
```

```
        }

        /* Obtain status bits from device */

        mask = nicptr->interruptMask;
        status = nicptr->interruptStatus & mask;

        /* Record status in ether struct */

        ethptr->interruptStatus = status;

        if (status == 0) {
                return;
        }

        sched_cntl(DEFER_START);

        if (status & IRQ_TX_PKTSENT) {  /* handle transmitter interrupt */
                ethptr->txirq++;
                txPackets(ethptr, nicptr);
        }

        if (status & IRQ_RX_PKTRECV) {  /* handle receiver interrupt    */
                ethptr->rxirq++;
                rxPackets(ethptr, nicptr);
        }

        /* Handle errors (transmit or receive overflow) */

        if (status & IRQ_RX_OVERFLOW) {
                /* Clear interrupt and restart processing */
                nicptr->rxStatus = RX_STAT_OVERFLOW;
                nicptr->rxControl = RX_CTRL_RXE;
                ethptr->errors++;
        }

        if ((status & IRQ_TX_UNDERFLOW) ||
                (status & IRQ_TX_BUSERR) || (status & IRQ_RX_BUSERR)) {
                panic("Catastrophic Ethernet error");
        }
        sched_cntl(DEFER_STOP);
        return;
}
```

Note that a transmit underflow or bus error should not occur unless the hardware malfunctions, and there is no way for the driver to correct the problem. Therefore, our driver code calls panic.

## 16.13 Ethernet Control Functions

The Ethernet driver supports three control functions: a caller can fetch the MAC address from the device, set the MAC address, and set *loopback mode*, which is used for testing. We usually think of an Ethernet MAC address as being hardwired into the device. At the lowest level, however, a manufacturer does not want to assign a permanent address until the hardware has been tested. Therefore, the example system is typical of small embedded designs: instead of being burned onto the Ethernet interface chip, the MAC address is loaded into non-volatile RAM. When it starts, the system must extract the address and load it into the device.

Because the underlying bus uses 32-bit transfers, the device divides a 48-bit MAC address into two pieces. To set the MAC address, *ethControl* must set both values. The code starts by picking up the first four bytes of the MAC address the user specifies, shifting each byte into its position within a 32-bit integer, and storing the result on the device. It then picks up the last two bytes of the MAC address, shifts them into position, and stores the result in the second integer on the device.

Obtaining a MAC address from the device works the opposite way from storing a MAC address. *EthControl* starts by obtaining an integer value from the device. It then shifts and masks each byte, which it stores in the array the caller specifies. Once the first four bytes have been stored, *ethControl* reads the second integer and extracts the final two bytes. File *ethControl.c* contains the code.

```
/* ethControl.c -  ethControl */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethControl - implement control function for an Ethernet device
 *------------------------------------------------------------------------
 */
devcall ethControl (
         struct dentry  *devptr,        /* entry in device switch table */
         int32  func,                   /* control function             */
         int32  arg1,                   /* argument 1, if needed        */
         int32  arg2                    /* argument 2, if needed        */
        )
{
        struct  ether   *ethptr;        /* ptr to control block         */
        struct  ag71xx  *nicptr;        /* ptr to device CSRs           */
```

```
byte    *macptr;                /* ptr to MAC address           */
uint32  temp;                   /* temporary                    */

ethptr = &ethertab[devptr->dvminor];
if (ethptr->csr == NULL) {
        return SYSERR;
}
nicptr = ethptr->csr;

switch (func) {

/* Program MAC address into card. */

case ETH_CTRL_SET_MAC:
        macptr = (byte *)arg1;

        temp  = ((uint32)macptr[0]) << 24;
        temp |= ((uint32)macptr[1]) << 16;
        temp |= ((uint32)macptr[2]) <<  8;
        temp |= ((uint32)macptr[3]) <<  0;
        nicptr->macAddr1 = temp;

        temp  = 0;
        temp  = ((uint32)macptr[4]) << 24;
        temp |= ((uint32)macptr[5]) << 16;
        nicptr->macAddr2 = temp;
        break;

/* Get MAC address from card */

case ETH_CTRL_GET_MAC:
        macptr = (byte *)arg1;

        temp = nicptr->macAddr1;
        macptr[0] = (temp >> 24) & 0xff;
        macptr[1] = (temp >> 16) & 0xff;
        macptr[2] = (temp >>  8) & 0xff;
        macptr[3] = (temp >>  0) & 0xff;

        temp = nicptr->macAddr2;
        macptr[4] = (temp >> 24) & 0xff;
        macptr[5] = (temp >> 16) & 0xff;
        break;

/* Set receiver mode */
```

```
        case ETH_CTRL_SET_LOOPBK:
                if (TRUE == (uint32)arg1) {
                        nicptr->macConfig1 |= MAC_CFG1_LOOPBACK;
                } else {
                        nicptr->macConfig1 &= ~MAC_CFG1_LOOPBACK;
                }
                break;
        default:
                return SYSERR;
    }
    return OK;
}
```

## 16.14 Perspective

DMA devices present an interesting irony to a programmer who must write a device driver. On the one hand, DMA hardware can be incredibly complex, and the documentation (which is called a *data sheet*) is often so difficult to understand that programmers find it impenetrable. Unlike a device with a few simple control and status registers, a DMA device requires a programmer to create complex data structures in memory and to communicate their location to the device. Furthermore, a programmer must understand exactly how and when the hardware sets response bits in the data structures and how the hardware interprets the requests that the operating system generates. On the other hand, once a programmer masters the documentation, the resulting driver code is usually smaller than the code for a non-DMA device. Thus, DMA devices have a steep learning curve, but offer the reward of both higher performance and smaller driver code.

## 16.15 Summary

A device that uses Direct Memory Access (DMA) can move an arbitrary block of data between the device and memory without using the CPU to fetch individual words of data. A DMA device typically uses a buffer ring in memory, where each node in the ring points to one buffer. Once the driver points the hardware to a node of the ring, the DMA engine performs the operation and moves to the next node on the ring automatically.

The chief advantage of a DMA device lies in lower overhead: the device only needs to interrupt the CPU once per block instead of once per byte or once per word. The driver code for a DMA device is simpler than the code for a conventional device because the driver does not need to perform low-level operations.

## EXERCISES

**16.1** The driver code uses array indexing to move from one node to the next. If the code is modified to follow a link instead of using array indexing, does the driver become more or less efficient?

**16.2** Read about Ethernet packets and find the minimum packet size. At 100 Mbps, how many packets can arrive per second?

**16.3** Build a test program that transmits Ethernet packets as fast as possible. How many large packets can you send per second? How many small packets?

**16.4** The current driver is complex and the code is somewhat difficult to read. Rewrite the code to use arrays for the transmit and receive rings. Allocate packet buffers statically.

# Chapter Contents

# 17

# A Minimal Internet Protocol Stack

*The lure of the distant and the difficult is deceptive. The great opportunity is where you are.*

— John Burroughs

## 17.1 Introduction

Because most embedded systems use a network to communicate, network protocol software has become a standard part of even small embedded operating systems. The previous chapter describes a basic Ethernet device driver that can send and receive packets. Although an Ethernet device can transfer packets, additional communication software is required to permit applications to communicate across the Internet. In particular, most systems use the *TCP/Internet Protocol Suite*, the protocols that define Internet communication. The protocols are organized into conceptual layers, and an implementation is known as a *protocol stack*.

A complete TCP/IP stack contains many protocols, and requires much more than a single chapter to describe. Therefore, this chapter describes a minimal implementation that is sufficiently powerful to support the remote disk and remote file systems covered later in the book. It provides a brief description without delving into the details of the protocols; the reader is referred to other texts from the author that explain the protocol suite and a full implementation.

## 17.2 Required Functionality

Our implementation of Internet protocols allows a process running on Xinu to communicate with an application running on a remote computer in the Internet (e.g., a PC, Mac, or Unix system, such as Linux or Solaris). It is possible to identify a remote computer and exchange messages with the computer. The system includes a timeout mechanism that allows a receiver to be informed if no message is received within a specified timeout.

In terms of protocols, our implementation supports:

- IP Internet Protocol
- UDP User Datagram Protocol
- ARP Address Resolution Protocol
- DHCP Dynamic Host Configuration Protocol
- ICMP Internet Control Message Protocol

*IP*. The *Internet Protocol* defines the format of an internet packet, which is known as a *datagram*. Each datagram is carried in the data area of an Ethernet frame. The Internet Protocol also defines the address format. Our implementation does not support IP options or features such as fragmentation (i.e., it is not a complete implementation). Packet forwarding follows the pattern used in most end systems: our IP software knows the computer's IP address, address mask for the local network, and a single *default router* address; if a destination is not on the local network, the packet is sent to the default router.

*UDP*. The *User Datagram Protocol* defines a set of 16-bit *port numbers* that an operating system uses to identify a specific application program. Communicating applications must agree on the port numbers they will use. Port numbers allow simultaneous communication without interference: an application can interact with one remote server while a second application interacts with another. Our software allows a process to specify a port number at run-time.

*ARP*. The *Address Resolution Protocol* provides two functions. Before another computer can send IP packets to our system, the computer must send an ARP packet that requests our Ethernet address and our system must respond with an ARP reply. Similarly, before our system can send IP packets to another computer, it first sends an ARP request to obtain the computer's Ethernet address, then uses the Ethernet address to send IP packets.

*DHCP*. The *Dynamic Host Configuration Protocol* provides a mechanism that a computer can use to obtain an IP address, an address mask for the network, and the IP address of a *default router*. The computer broadcasts a request, and a DHCP server running on the network sends a response. Usually, DHCP is invoked at startup because the information must be obtained before normal Internet communication is possible. Our implementation does not invoke DHCP immediately at startup. Instead, it waits until a process attempts to obtain a local IP address.

*ICMP*. The *Internet Control Message Protocol* provides error and informational messages that support IP. Our implementation only handles the two ICMP messages used by the *ping* program: *Echo Request* and *Echo Reply*. Because the code for ICMP is large, we describe the structure of the protocol software without showing all the details; the code is available on the web site for the text.†

## 17.3 Simultaneous Conversations, Timeouts, And Processes

How should protocol software be organized? How many processes are needed? A full stack includes many protocols and permits them to be used simultaneously. To accommodate simultaneous use, many implementations use multiple processes, with one process assigned to each protocol. Other implementations use software interrupts as an alternative to processes. As we will see, our minimal software can implement the set of protocols described above with only two extra processes. This section describes the process structure, and later sections of the chapter examine the code.

Why are processes needed? The answer lies in a technique that is fundamental to network protocol design. Known as *timeout-and-retransmission*, the technique handles situations where packets are lost (e.g., because the queue at a server overflows). To use timeout-and-retransmission, a protocol must be designed so the receiver sends a response to each message. When it transmits a message, a sender starts a timer. If the timer expires before a response arrives, the sender assumes that the message was lost and retransmits a second copy.

Our software uses an elegant design that has a single network input process, named *netin*.‡ The design uses function *recvtime* to handle all timeouts. That is, after transmitting a message, a sender calls *recvtime* to wait for a response. When a response arrives, the network input process, *netin*, sends a message to the waiting process, and *recvtime* returns the message. If the timer expires, *recvtime* returns value *TIMEOUT*. To make the system work, a sender must coordinate with the *netin* process. That is, before it transmits a message, a sender must store its process ID in a location that the *netin* process knows. For ARP messages, the process ID is stored in the ARP table entry for the address being resolved. For UDP messages, the process ID is stored with a packet queue in the UDP table entry for the port being used. Figure 17.1 illustrates how the *netin* process stores an incoming UDP packet in a UDP queue or extracts information from an incoming ARP packet and places the information in an ARP table entry. In either case, if a process is waiting, *netin* sends a message to the waiting process.

†URL: *xinu.cs.purdue.edu*

‡The code for *netin* can be found later in the chapter on page 348.

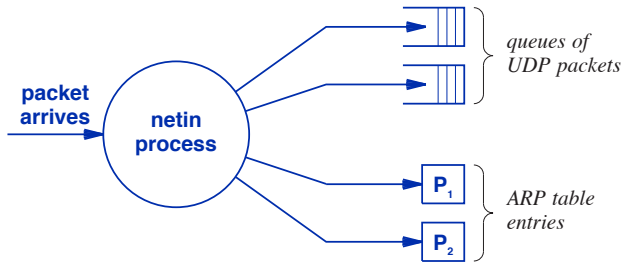



**Figure 17.1** The conceptual function of the *netin* process.

On output, our design allows application processes to invoke output functions. The only exception arises from ICMP echo: a separate ICMP output process is used to handle *ping* replies. The exception is required because we must decouple ICMP input and output, allowing the network input process to continue running while ICMP sends a reply. Decoupling is needed because an ICMP reply travels in an IP packet, and sending an IP packet may require an ARP exchange. For ARP to work, the network input process must continue to execute (i.e., must read and handle incoming ARP replies). Thus, if the network input process blocks waiting for an ARP response, the system will deadlock. Figure 17.2 illustrates the decoupling by showing how the *netin* process places outgoing ICMP packets in a queue for the ICMP output process.

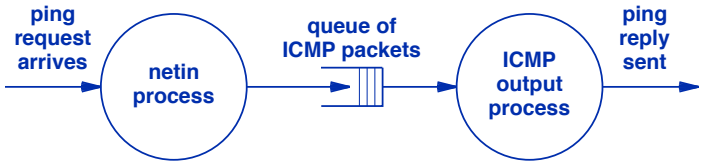



**Figure 17.2** The process structure used to handle *ping* replies.

## 17.4 ARP Functions

Before two computers on an Ethernet can communicate using the Internet Protocol, they must learn each other's Ethernet addresses. The protocol exchanges two messages: computer A broadcasts an *ARP request* that contains an IP address. Whichever computer on the network has the IP address in the request sends an *ARP response* that specifies its Ethernet address. When a response arrives, an entry is added to a table that is known as an *ARP cache*. The entry contains the remote computer's IP address and its Ethernet address. Subsequent transmissions to the same destination extract the information from the ARP cache without sending another request.

Our implementation stores ARP information in array *arpcache*. Struct *arpentry* defines the contents of each entry in the array to consist of: a state field (which specifies whether the entry is currently unused, being filled in, or already filled in), an IP address, the corresponding Ethernet address, and a process ID. If the entry is in the pending state, the process ID field contains the ID of the process that is waiting for the information to arrive. File *arp.h* defines the ARP packet format (when used on an Ethernet) and the format of an ARP cache entry.

```
/* arp.h */

/* Items related to ARP - definition of cache and the packet format */

#define ARP_HALEN       6               /* size of Ethernet MAC address */
#define ARP_PALEN       4               /* size of IP address           */

#define ARP_HTYPE       1               /* Ethernet hardware type       */
#define ARP_PTYPE       0x0800          /* IP protocol type             */

#define ARP_OP_REQ      1               /* Request op code              */
#define ARP_OP_RPLY     2               /* Reply op code                */

#define ARP_SIZ         16              /* number of entries in a cache */

#define ARP_RETRY       3               /* num. retries for ARP request */

#define ARP_TIMEOUT     200             /* retry timer in milliseconds  */

/* State of an ARP cache entry */

#define AR_FREE         0               /* slot is unused               */
#define AR_PENDING      1               /* resolution in progress       */
#define AR_RESOLVED     2               /* entry is valid               */

#pragma pack(2)
struct  arppacket {                     /* ARP packet for IP & Ethernet */
        byte    arp_ethdst[ETH_ADDR_LEN];/* Ethernet dest. MAC addr     */
        byte    arp_ethsrc[ETH_ADDR_LEN];/* Ethernet source MAC address */
        uint16  arp_ethtype;            /* Ethernet type field          */
        uint16  arp_htype;              /* ARP hardware type            */
        uint16  arp_ptype;              /* ARP protocol type            */
        byte    arp_hlen;               /* ARP hardware address length  */
        byte    arp_plen;               /* ARP protocol address length  */
        uint16  arp_op;                 /* ARP operation                */
        byte    arp_sndha[ARP_HALEN];   /* ARP sender's Ethernet addr.  */
        uint32  arp_sndpa;              /* ARP sender's IP address      */
```

```
        byte    arp_tarha[ARP_HALEN];   /* ARP target's Ethernet addr.  */
        uint32  arp_tarpa;              /* ARP target's IP address      */
};
#pragma pack()

struct  arpentry {                      /* entry in the ARP cache       */
        int32   arstate;                /* state of the entry           */
        uint32  arpaddr;                /* IP address of the entry      */
        pid32   arpid;                  /* waiting process or -1        */
        byte    arhaddr[ARP_HALEN];     /* Ethernet address of the entry*/
};

extern  struct  arpentry arpcache[];
```

ARP uses the same packet format for both requests and responses; a field in the header specifies the type as a request or response. In each case, the packet contains the sender's IP address and Ethernet address as well as the target's IP address and Ethernet address. In a request, the target's Ethernet address is unknown, so the field contains zeroes.

Our ARP software consists of four functions, *arp_init*, *arp_resolve*, *arp_in*, and *arp_alloc*. All four functions reside in a single source file, *arp.c*:

```
/* arp.c - arp_init, arp_resolve, arp_in, arp_alloc */

#include <xinu.h>

struct  arpentry  arpcache[ARP_SIZ];    /* ARP cache                    */
sid32   arpmutex;                       /* Mutual exclusion semaphore   */

/*------------------------------------------------------------------------
 * arp_init - initialize ARP mutex and cache
 *------------------------------------------------------------------------
 */
void    arp_init(void) {

        int32   i;                      /* ARP cache index              */

        arpmutex = semcreate(1);
        for (i=1; i<ARP_SIZ; i++) {     /* initialize cache to empty    */
                arpcache[i].arstate = AR_FREE;
        }
}

/*------------------------------------------------------------------------
```

```
 * arp_resolve - use ARP to resolve an IP address into an Ethernet address
 *------------------------------------------------------------------------
 */
status  arp_resolve (
         uint32 ipaddr,                 /* IP address to resolve        */
         byte   mac[ETH_ADDR_LEN]       /* array into which Ethernet    */
        )                               /*  address should be placed    */
{
        struct  arppacket apkt;         /* local packet buffer          */
        int32   i;                      /* index into arpcache          */
        int32   slot;                   /* ARP table slot to use        */
        struct  arpentry  *arptr;       /* ptr to ARP cache entry       */
        int32   msg;                    /* message returned by recvtime */
        byte    ethbcast[] = {0xff,0xff,0xff,0xff,0xff,0xff};

        if (ipaddr == IP_BCAST) {       /* set mac address to b-cast    */
                memcpy(mac, ethbcast, ETH_ADDR_LEN);
                return OK;
        }

        /* Insure only one process uses ARP at a time */

        wait(arpmutex);
        for (i=0; i<ARP_SIZ; i++) {
                arptr = &arpcache[i];
                if (arptr->arstate == AR_FREE) {
                        continue;
                }
                if (arptr->arpaddr == ipaddr) { /* adddress is in cache */
                        break;
                }
        }

        if (i < ARP_SIZ) {      /* entry was found */

                /* Only one request can be pending for an address */

                if (arptr->arstate == AR_PENDING) {
                        signal(arpmutex);
                        return SYSERR;
                }

                /* Entry is resolved - handle and return */

                memcpy(mac, arptr->arhaddr, ARP_HALEN);
```

```
        signal(arpmutex);
        return OK;
}

/* Must allocate a new cache entry for the request */

slot = arp_alloc();
if (slot == SYSERR) {
        signal(arpmutex);
        return SYSERR;
}
arptr = &arpcache[slot];
arptr->arstate = AR_PENDING;
arptr->arpaddr = ipaddr;
arptr->arpid = currpid;

/* Release ARP cache for others */

signal(arpmutex);

/* Hand-craft an ARP Request packet */

memcpy(apkt.arp_ethdst, ethbcast, ETH_ADDR_LEN);
memcpy(apkt.arp_ethsrc, NetData.ethaddr, ETH_ADDR_LEN);
apkt.arp_ethtype = ETH_ARP;       /* Packet type is ARP         */
apkt.arp_htype = ARP_HTYPE;       /* Hardware type is Ethernet  */
apkt.arp_ptype = ARP_PTYPE;       /* Protocol type is IP        */
apkt.arp_hlen = 0xff & ARP_HALEN; /* Ethernet MAC size in bytes */
apkt.arp_plen = 0xff & ARP_PALEN; /* IP address size in bytes   */
apkt.arp_op = 0xffff & ARP_OP_REQ;/* ARP type is Request        */
memcpy(apkt.arp_sndha, NetData.ethaddr, ARP_HALEN);
apkt.arp_sndpa = NetData.ipaddr;  /* Local IP address           */
memset(apkt.arp_tarha, '\0', ARP_HALEN); /* Target HA is unknown*/
apkt.arp_tarpa = ipaddr;          /* Target protocol address    */

/* Send the packet ARP_RETRY times and await response*/

msg = recvclr();
for (i=0; i<ARP_RETRY; i++) {
        write(ETHER0, (char *)&apkt, sizeof(struct arppacket));
        msg = recvtime(ARP_TIMEOUT);
        if (msg == TIMEOUT) {
                continue;
        } else if (msg == SYSERR) {
                return SYSERR;
```

```
                } else {        /* entry is resolved */
                        break;
                }
        }

        /* Verify that entry has not changed */

        if (arptr->arpaddr != ipaddr) {
                return SYSERR;
        }

        /* Either return hardware address or TIMEOUT indicator */

        if (i < ARP_RETRY) {
                memcpy(mac, arptr->arhaddr, ARP_HALEN);
                return OK;
        } else {
                arptr->arstate = AR_FREE;   /* invalidate cache entry */
                return TIMEOUT;
        }
}


/*------------------------------------------------------------------------
 * arp_in - handle an incoming ARP packet
 *------------------------------------------------------------------------
 */
void    arp_in (void) {                 /* currpkt points to the packet */

        struct  arppacket *pktptr;      /* ptr to incoming packet       */
        struct  arppacket apkt;         /* Local packet buffer          */
        int32   slot;                   /* slot in cache                */
        struct  arpentry  *arptr;       /* ptr to ARP cache entry       */
        bool8   found;                  /* is the sender's address in   */
                                        /*   the cache?                 */

        /* Insure only one process uses ARP at a time */

        wait(arpmutex);

        pktptr = (struct arppacket *)currpkt;

        /* Search cache for sender's IP address */

        found = FALSE;
```

```
for (slot=0; slot < ARP_SIZ; slot++) {
        arptr = &arpcache[slot];

        /* Ignore unless entry valid and address matches */

        if ( (arptr->arstate != AR_FREE) &&
             (arptr->arpaddr == pktptr->arp_sndpa) ) {
                found = TRUE;
                break;
        }
}

if (found) {    /* Update sender's hardware address */

        memcpy(arptr->arhaddr, pktptr->arp_sndha, ARP_HALEN);

        /* Handle entry that was pending */

        if (arptr->arstate == AR_PENDING) {
                arptr->arstate = AR_RESOLVED;

                /* Notify waiting process */

                send(arptr->arpid, OK);
        }
}

/* For an ARP reply, processing is complete */

if (pktptr->arp_op == ARP_OP_RPLY) {
        signal(arpmutex);
        return;
}

/* ARP request packet: if local machine is not  the target,     */
/*      processing is complete                                  */

if ((! NetData.ipvalid) || (pktptr->arp_tarpa!=NetData.ipaddr)) {
        signal(arpmutex);
        return;
}

/* Request has been sent to local machine: add sender's info    */
/*      to cache, if not already present                        */
```

```
        if (! found) {
                slot = arp_alloc();
                if (slot == SYSERR) {   /* cache overflow */
                        signal(arpmutex);
                        return;
                }
                arptr = &arpcache[slot];
                arptr->arstate = AR_RESOLVED;
                arptr->arpaddr = pktptr->arp_sndpa;
                memcpy(arptr->arhaddr, pktptr->arp_sndha, ARP_HALEN);
        }

        /* Hand-craft an ARP reply packet and send */

        memcpy(apkt.arp_ethdst, pktptr->arp_sndha, ARP_HALEN);
        memcpy(apkt.arp_ethsrc, NetData.ethaddr, ARP_HALEN);
        apkt.arp_ethtype= ETH_ARP;      /* Frame carries ARP    */
        apkt.arp_htype  = ARP_HTYPE;    /* Hardware is Ethernet */
        apkt.arp_ptype  = ARP_PTYPE;    /* Protocol is IP       */
        apkt.arp_hlen   = ARP_HALEN;    /* Ethernet address size*/
        apkt.arp_plen   = ARP_PALEN;    /* IP address size      */
        apkt.arp_op     = ARP_OP_RPLY;  /* Type is Reply        */

        /* Insert local Ethernet and IP address in sender fields */

        memcpy(apkt.arp_sndha, NetData.ethaddr, ARP_HALEN);
        apkt.arp_sndpa = NetData.ipaddr;

        /* Copy target Ethernet and IP addresses from request packet */

        memcpy(apkt.arp_tarha, pktptr->arp_sndha, ARP_HALEN);
        apkt.arp_tarpa = pktptr->arp_sndpa;

        /* Send the reply */

        write(ETHER0, (char *)&apkt, sizeof(struct  arppacket));
        signal(arpmutex);
        return;
}

/*------------------------------------------------------------------------
 * arp_alloc - find a free slot or kick out an entry to create one
 *------------------------------------------------------------------------
 */
```

```
int32   arp_alloc (void) {

        static  int32   nextslot = 0;   /* next slot to try             */
        int32   i;                      /* counts slots in the table    */
        int32   slot;                   /* slot that is selected        */

        /* Search for free slot starting at nextslot */

        for (i=0; i < ARP_SIZ; i++) {
                slot = nextslot++;
                if (nextslot >= ARP_SIZ) {
                        nextslot = 0;
                }
                if (arpcache[slot].arstate == AR_FREE) {
                        return slot;
                }
        }

        /* Search for resolved entry */

        slot = nextslot + 1;
        for (i=0; i < ARP_SIZ; i++) {
                if (slot >= ARP_SIZ) {
                        slot = 0;
                }
                if (arpcache[slot].arstate == AR_RESOLVED) {
                        return slot;
                }
        }

        /* All slots are pending */

        kprintf("ARP cache size exceeded\n\r");

        return SYSERR;
}
```

*Arp_init*. Function *arp_init* is called once when the system starts. It marks each entry in the ARP cache free and creates a mutual exclusion semaphore that insures only one process will attempt to change the ARP cache (e.g., insert an entry) at any time. Functions *arp_resolve* and *arp_in* are used to handle address lookup for outgoing IP packets and to process incoming ARP packets, respectively. The final function, *arp_alloc*, is called to allocate an entry in the table whenever a new item must be added.

*Arp_resolve*. Function *arp_resolve* is called when an IP packet is ready to be sent. *Arp_resolve* takes two arguments: the first specifies the IP address of a computer for which an Ethernet address is needed; the second is a pointer to an array that will hold the Ethernet address.

Although the code may seem complex, there are only three cases: the IP address is a broadcast address, the information is already in the ARP cache, or the information is not known. For an IP broadcast address, *arp_resolve* copies the Ethernet broadcast address into the array specified by the second argument. If the information is present in the cache, *arp_resolve* finds the correct entry, copies the Ethernet address from the entry into the caller's array, and returns to the caller without sending any packets over the network.

In the case where the requested mapping is not in the cache, *arp_resolve* must send packets over the network to obtain the information. The exchange involves sending a request and waiting for a reply. *Arp_resolve* creates an entry in the table, marks the entry *AR_PENDING*, forms an ARP request packet, broadcasts the packet on the local network, and then waits for a reply. As discussed above, *arp_resolve* uses *recvtime* to wait. The call to *recvtime* will return if a response arrives or the timer expires, whichever occurs first. In the next section, we will describe how an incoming packet is processed and how a message is sent to a waiting process.

The code is more complex than we have described because *arp_resolve* does not merely give up if a timeout occurs. Instead, our implementation is designed to retry the operation: it sends a request and waits for a reply *ARP_RETRY* times before it returns *TIMEOUT* to the caller.

*Arp_in*. The second major ARP function runs when an incoming ARP packet arrives. The *netin* process examines the type field in each incoming Ethernet packet. If it finds the ARP packet type (*0x806*), *netin* calls function *arp_in* to handle the packet. *Arp_in* must handle two cases: either the packet is a request that was initiated by another computer or it is a reply, possibly to a request that we have sent.

The protocol specifies that when either type of packet arrives, ARP must examine the sender's information (IP address and Ethernet address), and update the local cache accordingly. If a process is waiting for the reply, *arp_in* sends a message to the process.

Because an ARP request is broadcast, all computers on the network receive each request. Therefore, after it updates the sender's information, *arp_in* checks the target IP address in a request to determine whether the request is for the local system or some other computer on the network. If the request is for another computer, *arp_in* returns without taking further action. If the target IP address in the incoming request matches the IP address of the local system, *arp_in* sends an ARP reply. *Arp_in* forms a reply in variable *apkt*. Once all fields of the packet have been filled in, the code calls *write* on the Ethernet device to transmit the reply back to the requester.

## 17.5 Definition Of A Network Packet

Our minimal implementation of network protocols combines IP, UDP, ICMP, and Ethernet. That is, we use a single data structure, named *netpacket*, to describe an Ethernet packet that carries an IP datagram which either carries a UDP message or an ICMP message. File *net.h* defines *netpacket* as well as other constants and data structures.

```
/* net.h */

/* Constants used in the networking code */

#define ETH_ARP     0x0806              /* Ethernet type for ARP        */
#define ETH_IP      0x0800              /* Ethernet type for IP         */

#define IP_BCAST    0xffffffff          /* IP local broadcast address   */
#define IP_THIS     0xffffffff          /* "this host" src IP address   */

#define IP_ICMP     1                   /* ICMP protocol type for IP    */
#define IP_UDP      17                  /* UDP protocol type for IP     */

#define IP_ASIZE    4                   /* bytes in an IP address       */
#define IP_HDR_LEN  20                  /* bytes in an IP header        */

/* Format of an Ethernet packet carrying IPv4 and UDP */

#pragma pack(2)
struct  netpacket       {
        byte    net_ethdst[ETH_ADDR_LEN];/* Ethernet dest. MAC address  */
        byte    net_ethsrc[ETH_ADDR_LEN];/* Ethernet source MAC address */
        uint16  net_ethtype;            /* Ethernet type field          */
        byte    net_ipvh;               /* IP version and hdr length    */
        byte    net_iptos;              /* IP type of service           */
        uint16  net_iplen;              /* IP total packet length       */
        uint16  net_ipid;               /* IP datagram ID               */
        uint16  net_ipfrag;             /* IP flags & fragment offset   */
        byte    net_ipttl;              /* IP time-to-live              */
        byte    net_ipproto;            /* IP protocol (actually type)  */
        uint16  net_ipcksum;            /* IP checksum                  */
        uint32  net_ipsrc;              /* IP source address            */
        uint32  net_ipdst;              /* IP destination address       */
        union {
         struct {
          uint16        net_udpsport;   /* UDP source protocol port     */
          uint16        net_udpdport;   /* UDP destination protocol port*/
```

```
            uint16        net_udplen;     /* UDP total length             */
            uint16        net_udpcksum;   /* UDP checksum                 */
            byte  net_udpdata[1500-42];/* UDP payload (1500-above)*/
           };
           struct  {
            byte          net_ictype;     /* ICMP message type            */
            byte          net_iccode;     /* ICMP code field (0 for ping) */
            uint16        net_iccksum;    /* ICMP message checksum        */
            uint16        net_icident;    /* ICMP identifier              */
            uint16        net_icseq;      /* ICMP sequence number         */
            byte          net_icdata[1500-42];/* ICMP payload (1500-above)*/
           };
          };
};
#pragma pack()

extern  struct  netpacket *currpkt;     /* ptr to current input packet  */
extern  bpid32  netbufpool;             /* ID of net packet buffer pool */

struct  network {
        uint32  ipaddr;                 /* IP address                   */
        uint32  addrmask;               /* Subnet mask                  */
        uint32  routeraddr;             /* Address of default router    */
        bool8   ipvalid;                /* Is IP address valid yet?     */
        byte    ethaddr[ETH_ADDR_LEN];  /* Ethernet address             */
};

extern  struct  network NetData;        /* Local network interface      */
```

## 17.6 The Network Input Process

At startup, Xinu creates the network input process, *netin*. Therefore, *netin* starts running before any application processes. After it creates a buffer pool and initializes global variables, *netin* calls initialization functions *arp_init*, *udp_init*, and *icmp_init*. It then allocates an initial network buffer and enters an infinite loop that repeatedly reads and processes packets. When it reads a packet from the Ethernet, *netin* uses the Ethernet *type field* in the packet to determine whether the Ethernet packet is carrying an ARP message or an IP datagram. In the case of ARP, *netin* calls *arp_in* to handle the packet. In the case of an IP datagram, *netin* verifies that the IP header checksum is valid, verifies that the destination IP address matches the broadcast address or the local address, and then uses the type field in the IP header to verify that the datagram is carrying UDP or ICMP. If any of the tests fail, *netin* goes on to the next packet. If the tests indicate that the information is valid, *netin* calls *icmp_in* or *udp_in* to process the packet. File *netin.c* contains the code.

```
/* netin.c - netin */

#include <xinu.h>

bpid32  netbufpool;                     /* ID of network buffer pool    */

struct  netpacket *currpkt;             /* packet buffer being used now */

struct  network NetData;                /* local network interface      */

/*------------------------------------------------------------------------
 * netin - continuously read the next incoming packet and handle it
 *------------------------------------------------------------------------
 */

process netin(void) {

        status  retval;                 /* return value from function   */

        netbufpool = mkbufpool(PACKLEN, UDP_SLOTS * UDP_QSIZ +
                                ICMP_SLOTS * ICMP_QSIZ + ICMP_OQSIZ + 1);
        if (netbufpool == SYSERR) {
                kprintf("Cannot allocate network buffer pool");
                kill(getpid());
        }

        /* Copy Ethernet address to global variable */

        control(ETHER0, ETH_CTRL_GET_MAC, (int32)NetData.ethaddr, 0);

        /* Indicate that IP address, mask, and router are not yet valid */

        NetData.ipvalid = FALSE;

        NetData.ipaddr = 0;
        NetData.addrmask = 0;
        NetData.routeraddr = 0;

        /* Initialize ARP cache */

        arp_init();

        /* Initialize UDP table */

        udp_init();
```

```
/* Initialize ICMP table */

icmp_init();

currpkt = (struct netpacket *)getbuf(netbufpool);

/* Do forever: read packets from the network and process */

while(1) {
    retval = read(ETHER0, (char *)currpkt, PACKLEN);
    if (retval == SYSERR) {
        panic("Ethernet read error");
    }

    /* Demultiplex on Ethernet type */

    switch (currpkt->net_ethtype) {

        case ETH_ARP:
                arp_in();               /* Handle an ARP packet */
                continue;

        case ETH_IP:
                if (ipcksum(currpkt) != 0) {
                        kprintf("checksum failed\n\r");
                        continue;
                }

                if (currpkt->net_ipvh != 0x45) {
                        kprintf("version failed\n\r");
                        continue;
                }

                if ( (currpkt->net_ipdst != IP_BCAST) &&
                     (NetData.ipvalid) &&
                     (currpkt->net_ipdst != NetData.ipaddr) ) {
                        continue;
                }

                /* Demultiplex ICMP or UDP and ignore others */

                if (currpkt->net_ipproto == IP_ICMP) {
                        icmp_in();      /* Handle an ICMP packet*/
```

```
                        } else if (currpkt->net_ipproto == IP_UDP) {
                                udp_in();       /* Handle a UDP packet  */
                        }
                        continue;

                default:        /* Ignore all other Ethernet types */
                        continue;
                }
        }
}


/*------------------------------------------------------------------------
 * ipcksum - compute the IP checksum for a packet
 *------------------------------------------------------------------------
 */

uint16  ipcksum(
         struct  netpacket *pkt         /* ptr to a packet              */
        )
{
        uint16  *hptr;                  /* ptr to 16-bit header values  */
        int32   i;                      /* counts 16-bit values in hdr  */
        uint32  cksum;                  /* computed value of checksum   */

        hptr= (uint16 *) &pkt->net_ipvh;
        cksum = 0;
        for (i=0; i<10; i++) {
                cksum += (uint32) *hptr++;
        }
        cksum += (cksum >> 16);
        cksum = 0xffff & ~cksum;
        if (cksum == 0xffff) {
                cksum = 0;
        }
        return (uint16) (0xffff & cksum);
}
```

Our implementation of *netin* relies on a global variable, *currpkt* that always points to the packet currently being processed. That is, *currpkt* points to a buffer that is used for the current packet. Before the loop starts, *netin* calls *getbuf* to obtain a buffer, and assigns the buffer address to *currpkt*. On each iteration, *netin* reads a packet into the current buffer. Thus, when *netin* calls either *arp_in* or *udp_in*, *currpkt* points to the packet that should be processed. *Arp_in* extracts information from the packet, and

leaves *currpkt* pointing to a buffer that can be reused. In the case of *udp_in*, the incoming packet may need to be enqueued in one of the UDP table entries. If it does enqueue the current packet, *udp_in* allocates a new buffer and assigns the address of the buffer to *currpkt* before returning to *netin*. The system is designed to have enough network buffers to allow network processing to continue even if all UDP queues are full.

## 17.7 Definition Of The UDP Table

UDP maintains a table that specifies the set of UDP endpoints that are currently in use. Each endpoint consists of an IP address and a UDP port number. An entry in the table has four fields that specify two endpoint pairs, one for a remote computer and one for the local computer.

To act as a server that can receive a packet from an arbitrary remote computer, a process allocates a table entry, fills in the local endpoint information, and leaves the remote endpoint unspecified. To act as a client that communicates with a specific remote computer, a process allocates a table entry and fills in both the local and remote endpoint information.

In addition to endpoint information, each entry in the UDP table contains a queue of packets that have arrived from the remote system (i.e., packets where the endpoints specified in the packet match those in the table entry). Each entry in the UDP table is described by struct *udpentry*; file *udp.h* defines the structure as well as associated symbolic constants.

```
/* udp.h - declarations pertaining to User Datagram Protocol (UDP) */

#define UDP_SLOTS       6               /* num. of open UDP endpoints   */
#define UDP_QSIZ        8               /* packets enqueued per endpoint*/

#define UDP_DHCP_CPORT  68              /* port number for DHCP client  */
#define UDP_DHCP_SPORT  67              /* port number for DHCP server  */


/* Constants for the state of an entry */

#define UDP_FREE        0               /* entry is unused              */
#define UDP_USED        1               /* entry is being used          */
#define UDP_RECV        2               /* entry has a process waiting  */

#define UDP_HDR_LEN     8               /* bytes in a UDP header        */

struct  udpentry {                      /* entry in the UDP endpoint tbl*/
        int32   udstate;                /* state of entry: free/used    */
        uint32  udremip;                /* remote IP address (zero      */
                                        /*   means "don't care")        */
        uint32  udlocip;                /* local IP address             */
        uint16  udremport;              /* remote protocol port number  */
        uint16  udlocport;              /* local protocol port number   */
        int32   udhead;                 /* index of next packet to read */
        int32   udtail;                 /* index of next slot to insert */
        int32   udcount;                /* count of packets enqueued    */
        pid32   udpid;                  /* ID of waiting process        */
        struct  netpacket *udqueue[UDP_QSIZ];/* circular packet queue   */
};

extern  struct  udpentry udptab[];      /* table of UDP endpoints       */
```

## 17.8 UDP Functions

In our system, applications use UDP for all communication. Therefore, the UDP interface is designed to allow an application to send and receive UDP messages and to act as either a client or a server. Our UDP software includes seven functions: *udp_init*, *udp_in*, *udp_register*, *udp_recv*, *udp_recvaddr*, *udp_send*, and *udp_release*. The functions are collected into a single file, *udp.c*. Following the file, the text describes each UDP function.

```
/* udp.c - udp_init udp_in udp_register udp_recv udp_recvaddr udp_send  */
/*                       udp_release                                    */

#include <xinu.h>

struct  udpentry udptab[UDP_SLOTS];             /* table of UDP endpts  */

/*------------------------------------------------------------------------
 * udp_init - initialize UDP endpoint table
 *------------------------------------------------------------------------
 */
void    udp_init(void) {

        int32   i;                      /* table index */

        for(i=0; i<UDP_SLOTS; i++) {
                udptab[i].udstate = UDP_FREE;
        }

        return;
}

/*------------------------------------------------------------------------
 * udp_in - handle an incoming UDP packet
 *------------------------------------------------------------------------
 */
void    udp_in(void) {                  /* currpkt points to the packet */

        int32   i;                      /* index into udptab            */
        struct  udpentry *udptr;        /* pointer to udptab entry      */

        for (i=0; i<UDP_SLOTS; i++) {
            udptr = &udptab[i];
            if ( (udptr->udstate != UDP_FREE) &&
                 (currpkt->net_udpdport == udptr->udlocport)  &&
                    ((udptr->udremport == 0) ||
                        (currpkt->net_udpsport == udptr->udremport)) &&
                 (  ((udptr->udremip==0)     ||
                        (currpkt->net_ipsrc == udptr->udremip)))    ) {

                /* Entry matches incoming packet */

                if (udptr->udcount < UDP_QSIZ) {
                        udptr->udcount++;
                        udptr->udqueue[udptr->udtail++] = currpkt;
```

```
                        if (udptr->udtail >= UDP_QSIZ) {
                                udptr->udtail = 0;
                        }
                        currpkt = (struct netpacket *)getbuf(netbufpool);
                        if (udptr->udstate == UDP_RECV) {
                                udptr->udstate = UDP_USED;
                                send (udptr->udpid, OK);
                        }
                        return;
                }
            }
        }

        /* no match - simply discard packet */

        return;
}

/*------------------------------------------------------------------------
 * udp_register - register a remote (IP,port) and local port to receive
 *                incoming UDP messages from the specified remote site
 *                sent to a specific local port
 *------------------------------------------------------------------------
 */
status  udp_register (
         uint32 remip,                  /* remote IP address or zero    */
         uint16 remport,                /* remote UDP protocol port     */
         uint16 locport                 /* local UDP protocol port      */
        )
{
        int32   i;                      /* index into udptab            */
        struct  udpentry *udptr;        /* pointer to udptab entry      */

        /* See if request already registered */

        for (i=0; i<UDP_SLOTS; i++) {
                udptr = &udptab[i];
                if (udptr->udstate == UDP_FREE) {
                        continue;
                }
                if ((remport == udptr->udremport) &&
                        (locport == udptr->udlocport) &&
                        (remip   == udptr->udremip  ) ) {

                        /* Entry in table matches request */
```

```
                        return SYSERR;
                }
        }

        /* Find a free slot and allocate it */

        for (i=0; i<UDP_SLOTS; i++) {
                udptr = &udptab[i];
                if (udptr->udstate == UDP_FREE) {
                        udptr->udstate = UDP_USED;
                        udptr->udlocport = locport;
                        udptr->udremport = remport;
                        udptr->udremip = remip;
                        udptr->udcount = 0;
                        udptr->udhead = udptr->udtail = 0;
                        udptr->udpid = -1;
                        return OK;
                }
        }

        return SYSERR;
}

/*------------------------------------------------------------------------
 * udp_recv - receive a UDP packet
 *------------------------------------------------------------------------
 */
int32   udp_recv (
         uint32 remip,                  /* remote IP address or zero    */
         uint16 remport,                /* remote UDP protocol port     */
         uint16 locport,                /* local UDP protocol port      */
         char   *buff,                  /* buffer to hold UDP data      */
         int32  len,                    /* length of buffer             */
         uint32 timeout                 /* read timeout in msec         */
        )
{
        intmask mask;                   /* interrupt mask               */
        int32   i;                      /* index into udptab            */
        struct  udpentry *udptr;        /* pointer to udptab entry      */
        umsg32  msg;                    /* message from recvtime()      */
        struct  netpacket *pkt;         /* ptr to packet being read     */
        int32   msglen;                 /* length of UDP data in packet */
        char    *udataptr;              /* pointer to UDP data          */
```

```
mask = disable();
for (i=0; i<UDP_SLOTS; i++) {
        udptr = &udptab[i];
        if ((remport == udptr->udremport) &&
                (locport == udptr->udlocport) &&
                (remip   == udptr->udremip  ) ) {

                /* Entry in table matches request */

                break;
        }
}

if (i >= UDP_SLOTS) {
        restore(mask);
        return SYSERR;
}

if (udptr->udcount == 0) {              /* No packet is waiting */
        udptr->udstate = UDP_RECV;
        udptr->udpid = currpid;
        msg = recvclr();
        msg = recvtime(timeout);        /* Wait for a packet    */
        udptr->udstate = UDP_USED;
        if (msg == TIMEOUT) {
                restore(mask);
                return TIMEOUT;
        } else if (msg != OK) {
                restore(mask);
                return SYSERR;
        }
}

/* Packet has arrived -- dequeue it */

pkt = udptr->udqueue[udptr->udhead++];
if (udptr->udhead >= UDP_SLOTS) {
        udptr->udhead = 0;
}
udptr->udcount--;

/* Copy UDP data from packet into caller's buffer */

msglen = pkt->net_udplen - UDP_HDR_LEN;
udataptr = (char *)pkt->net_udpdata;
```

```
        for (i=0; i<msglen; i++) {
                if (i >= len) {
                        break;
                }
                *buff++ = *udataptr++;
        }
        freebuf((char *)pkt);
        restore(mask);
        return i;
}

/*------------------------------------------------------------------------
 * udp_recvaddr - receive a UDP packet and record the sender's address
 *------------------------------------------------------------------------
 */
int32   udp_recvaddr (
         uint32 *remip,                 /* loc to record remote IP addr.*/
         uint16 *remport,               /* loc to record remote port    */
         uint16 locport,                /* local UDP protocol port      */
         char   *buff,                  /* buffer to hold UDP data      */
         int32  len,                    /* length of buffer             */
         uint32 timeout                 /* read timeout in msec         */
        )
{
        intmask mask;                   /* interrupt mask               */
        int32   i;                      /* index into udptab            */
        struct  udpentry *udptr;        /* pointer to udptab entry      */
        umsg32  msg;                    /* message from recvtime()      */
        struct  netpacket *pkt;         /* ptr to packet being read     */
        int32   msglen;                 /* length of UDP data in packet */
        char    *udataptr;              /* pointer to UDP data          */

        mask = disable();
        for (i=0; i<UDP_SLOTS; i++) {
                udptr = &udptab[i];
                if ( (udptr->udremip == 0 ) &&
                        (locport == udptr->udlocport) ) {

                        /* Entry in table matches request */
                        break;
                }
        }

        if (i >= UDP_SLOTS) {
                restore(mask);
```

```
                return SYSERR;
        }

        if (udptr->udcount == 0) {      /* no packet is waiting */
                udptr->udstate = UDP_RECV;
                udptr->udpid = currpid;
                msg = recvclr();
                msg = recvtime(timeout);        /* wait for packet */
                udptr->udstate = UDP_USED;
                if (msg == TIMEOUT) {
                        restore(mask);
                        return TIMEOUT;
                } else if (msg != OK) {
                        restore(mask);
                        return SYSERR;
                }
        }

        /* Packet has arrived -- dequeue it */

        pkt = udptr->udqueue[udptr->udhead++];
        if (udptr->udhead >= UDP_SLOTS) {
                udptr->udhead = 0;
        }
        udptr->udcount--;

        /* Record sender's IP address and UDP port number */

        *remip = pkt->net_ipsrc;
        *remport = pkt->net_udpsport;

        /* Copy UDP data from packet into caller's buffer */

        msglen = pkt->net_udplen - UDP_HDR_LEN;
        udataptr = (char *)pkt->net_udpdata;
        for (i=0; i<msglen; i++) {
                if (i >= len) {
                        break;
                }
                *buff++ = *udataptr++;
        }
        freebuf((char *)pkt);
        restore(mask);
        return i;
}
```

```
/*------------------------------------------------------------------------
 * udp_send - send a UDP packet
 *------------------------------------------------------------------------
 */
status  udp_send (
         uint32 remip,                  /* remote IP address or IP_BCAST*/
                                        /*  for a local broadcast       */
         uint16 remport,                /* remote UDP protocol port     */
         uint32 locip,                  /* local IP address             */
         uint16 locport,                /* local UDP protocol port      */
         char   *buff,                  /* buffer of UDP data           */
         int32  len                     /* length of data in buffer     */
        )
{
        struct  netpacket pkt;          /* local packet buffer          */
        int32   pktlen;                 /* total packet length          */
        static  uint16 ident = 1;       /* datagram IDENT field         */
        char    *udataptr;              /* pointer to UDP data          */
        byte    ethbcast[] = {0xff,0xff,0xff,0xff,0xff,0xff};

        /* Compute packet length as UDP data size + fixed header size   */

        pktlen = ((char *)pkt.net_udpdata - (char *)&pkt) + len;

        /* Create UDP packet in pkt */

        memcpy(pkt.net_ethsrc, NetData.ethaddr, ETH_ADDR_LEN);
        pkt.net_ethtype = 0x800;        /* Type is IP */
        pkt.net_ipvh = 0x45;            /* IP version and hdr length    */
        pkt.net_iptos = 0x00;           /* Type of service              */
        pkt.net_iplen= pktlen - ETH_HDR_LEN;/* total IP datagram length */
        pkt.net_ipid = ident++;         /* datagram gets next IDENT     */
        pkt.net_ipfrag = 0x0000;        /* IP flags & fragment offset   */
        pkt.net_ipttl = 0xff;           /* IP time-to-live              */
        pkt.net_ipproto = IP_UDP;       /* datagram carries UDP         */
        pkt.net_ipcksum = 0x0000;       /* initial checksum             */
        pkt.net_ipsrc = locip;          /* IP source address            */
        pkt.net_ipdst = remip;          /* IP destination address       */

        /* compute IP header checksum */
        pkt.net_ipcksum = 0xffff & ipcksum(&pkt);

        pkt.net_udpsport = locport;     /* local UDP protocol port      */
        pkt.net_udpdport = remport;     /* remote UDP protocol port     */
```

```
        pkt.net_udplen = (uint16)(UDP_HDR_LEN+len); /* UDP length       */
        pkt.net_udpcksum = 0x0000;      /* ignore UDP checksum          */
        udataptr = (char *) pkt.net_udpdata;
        for (; len>0; len--) {
                *udataptr++ = *buff++;
        }

        /* Set MAC address in packet, using ARP if needed */

        if (remip == IP_BCAST) {        /* set mac address to b-cast    */
                memcpy(pkt.net_ethdst, ethbcast, ETH_ADDR_LEN);

        /* If destination isn't on the same subnet, send to router      */

        } else if ((locip & NetData.addrmask)
                        != (remip & NetData.addrmask)) {
                if (arp_resolve(NetData.routeraddr, pkt.net_ethdst)!=OK) {
                    kprintf("udp_send: cannot resolve router %08x\n\r",
                                NetData.routeraddr);
                    return SYSERR;
                }
        } else {
                /* Destination is on local subnet -  get MAC address */

                if (arp_resolve(remip, pkt.net_ethdst) != OK) {
                        kprintf("udp_send: cannot resolve %08x\n\r",remip);
                        return SYSERR;
                }
        }

        write(ETHER0, (char *)&pkt, pktlen);
        return OK;
}

/*------------------------------------------------------------------------
 * udp_release - release a previously-registered remote IP, remote
 *                      port, and local port (exact match required)
 *------------------------------------------------------------------------
 */
status  udp_release (
         uint32 remip,                  /* remote IP address or zero    */
         uint16 remport,                /* remote UDP protocol port     */
         uint16 locport                 /* local UDP protocol port      */
        )
{
```

```
        int32   i;                      /* index into udptab            */
        struct  udpentry *udptr;        /* pointer to udptab entry      */
        struct  netpacket *pkt;         /* ptr to packet being read     */

        for (i=0; i<UDP_SLOTS; i++) {
                udptr = &udptab[i];
                if (udptr->udstate != UDP_USED) {
                        continue;
                }
                if ((remport == udptr->udremport) &&
                        (locport == udptr->udlocport) &&
                        (remip   == udptr->udremip  ) ) {

                        /* Entry in table matches */

                        sched_cntl(DEFER_START);
                        while (udptr->udcount > 0) {
                                pkt = udptr->udqueue[udptr->udhead++];
                                if (udptr->udhead >= UDP_SLOTS) {
                                        udptr->udhead = 0;
                                }
                                freebuf((char *)pkt);
                                udptr->udcount--;
                        }
                        udptr->udstate = UDP_FREE;
                        sched_cntl(DEFER_STOP);
                        return OK;
                }
        }
        return SYSERR;
}
```

*Udp_init*. The initialization function is easiest to understand. The system calls *udp_init* once during startup, and *udp_init* sets the state of each entry in the UDP table to indicate that the entry is unused.

*Udp_in*. The *netin* process calls function *udp_in* when a packet arrives carrying a UDP message. Global pointer *currpkt* points to the incoming packet. *Udp_in* searches the UDP table to see if an entry in the table matches the port numbers and IP addresses in the current packet. If there is no match, the incoming packet is dropped — *udp_in* simply returns, which means *netin* will use the same buffer to read the next packet. If a match does occur, *udp_in* inserts the incoming packet in the queue associated with the table entry. If the queue is full, *udp_in* merely returns, which means the packet will be dropped and the buffer will be used to read the next packet. When it inserts a packet in the queue, *udp_in* checks to see if a process is waiting for a packet to arrive (state

*UDP_RECV*), and sends a message to the waiting process. Note that at any time, only one process can be waiting for an entry in the table; if multiple processes need to use an entry to communicate, they must coordinate among themselves.

*Udp_register*. Before it can use UDP to communicate, an application must call *udp_register* to specify that it expects to receive incoming packets sent to a specific protocol port. The application can act as a client by specifying a remote IP address or can act as a server to accept packets from an arbitrary sender. *Udp_register* allocates an entry in the UDP table, records the remote and local protocol port and IP address information in the entry, and creates a queue to hold incoming packets.

*Udp_recv*. Once a local port number has been registered, an application can call *udp_recv* to extract a packet from a table entry. Arguments to the call specify a remote port and IP address as well as a local port. The three items specified in a call of *udp_recv* must match the items in a table entry (i.e., the combination of local and remote port and IP address must have been registered previously). *Udp_recv* uses the same paradigm as ARP. If no packet is waiting (i.e., the queue for the entry is empty), *udp_recv* blocks and waits for the amount of time specified by the last argument. When a UDP packet arrives, *netin* calls *udp_in*. The code in *udp_in* finds the appropriate entry in the UDP table, and if an application process is waiting, sends a message to the waiting process. Thus, if a packet arrives within the specified time, *udp_recv* copies the UDP data to the caller's buffer and returns the length. If the timer expires before a packet arrives, *udp_recv* returns *TIMEOUT*.

*Udp_recvaddr*. When it acts as a server, a process must learn the address of the client that contacted it. A server process calls *udp_recvaddr*, which acts like *udp_recv* except that the call returns both an incoming packet and the address of the sender. The server can use the address to send a reply.

*Udp_send*. A process calls *udp_send* to transmit a UDP message. Arguments specify the remote and local protocol port numbers for the packet, the remote and local IP addresses, the address of a message in memory, and the length of the message. *Udp_send* creates an Ethernet packet that contains an IP datagram carrying the specified UDP message. Care must be taken to use valid addresses and port numbers because *udp_send* merely copies information into the packet without checking that the information is valid.

*Udp_release*. Once a process has finished using an UDP endpoint, the process can call *udp_release* to release the table entry. If packets are enqueued in an entry, *udp_release* returns each to the buffer pool before marking the table entry free.

## 17.9 Internet Control Message Protocol

Our implementation of ICMP only handles the two message types used by the *ping* program: *ICMP Echo Request* and *ICMP Echo Reply*. Despite the restriction on message types, the code contains seven major functions: *icmp_init*, *icmp_in*, *icmp_out*, *icmp_register*, *icmp_send*, *icmp_recv*, and *icmp_release*.

As with other parts of the protocol stack, the network input function initializes ICMP by calling *icmp_init*. The network input process calls *icmp_in* when an ICMP packet arrives. An application process calls *icmp_register* to register its use of a remote IP address, then uses *icmp_send* to send a *ping* request and *icmp_recv* to receive a reply. Finally, once it has finished, the application calls *icmp_release* to release the remote IP address and allow other processes to use it.

Although the ICMP code is not shown,† the functions follow the same general structure as the UDP functions. A trick is used to associate *ping* replies with requests: the identification field in an outgoing *ping* packet is an index in the *ping* table. When a reply arrives, the reply contains the same identification, which *icmp_in* uses as an index into the array. Thus, unlike UDP, the ICMP code never needs to search the table. Of course, the identification field alone is not sufficient: once a table entry has been identified, *icmp_in* verifies that the IP source address in the reply matches the IP address in the entry.

*Icmp_out* runs as a separate process. Given an ICMP message to send, *icmp_out* uses ARP to resolve the IP address of the destination, and sends the packet over the Ethernet. Recall from the discussion above that a separate process is needed to insure that *netin* can continue to run even if ICMP output requires the *icmp_out* process to block waiting for an ARP reply.

## 17.10 Dynamic Host Configuration Protocol

When it boots, a computer must obtain its IP address and the IP address of a default router. The protocol used to obtain information at startup is known as the *Dynamic Host Configuration Protocol* (*DHCP*). Although a DHCP packet contains many fields, the basic packet exchange is straightforward. A computer, known as a *host*, broadcasts a DHCP *Discover* message. A DHCP server on the local network replies by sending a DHCP *Offer* message that contains an IP address for the host, a 32-bit subnet mask for the local network, and the address of a default router.

Instead of engaging in a DHCP exchange when the system boots, our code waits until an IP address is needed. An application calls *getlocalip* to obtain the local IP address. If the IP address has been fetched previously, the code merely returns the value. If the host's IP address is unknown, *getlocalip* uses DHCP to obtain the address. The code starts by creating and sending a DHCP *Discover* message. It then uses *udp_recv* to wait for a reply.

File *dhcp.h* defines the structure of a DHCP message. The entire DHCP message will be carried in the payload of a UDP message, which is carried in an IP datagram, which is carried in an Ethernet packet.

†The code can be obtained from the website *xinu.cs.purdue.edu*

```
/* dhcp.h - Definitions related to DHCP */

#define DHCP

#pragma pack(2)
struct  dhcpmsg {
        byte    dc_bop;                 /* DHCP bootp op 1=req 2=reply  */
        byte    dc_htype;               /* DHCP hardware type           */
        byte    dc_hlen;                /* DHCP hardware address length */
        byte    dc_hops;                /* DHCP hop count               */
        uint32  dc_xid;                 /* DHCP xid                     */
        uint16  dc_secs;                /* DHCP seconds                 */
        uint16  dc_flags;               /* DHCP flags                   */
        uint32  dc_cip;                 /* DHCP client IP address       */
        uint32  dc_yip;                 /* DHCP your IP address         */
        uint32  dc_sip;                 /* DHCP server IP address       */
        uint32  dc_gip;                 /* DHCP gateway IP address      */
        byte    dc_chaddr[16];          /* DHCP client hardware address */
        byte    dc_bootp[192];          /* DHCP bootp area (zero)       */
        uint32  dc_cookie;              /* DHCP cookie                  */
        byte    dc_opt[1024];           /* DHCP options area (large     */
                                        /*  enough to hold more than    */
                                        /*  reasonable options          */
};
#pragma pack()

extern  struct  netpacket *currpkt;     /* ptr to current input packet  */
extern  bpid32  netbufpool;             /* ID of net packet buffer pool */

#define PACKLEN sizeof(struct netpacket)

extern  uint32  myipaddr;               /* IP address of computer       */
```

If the local IP address has not been initialized, function *getlocalip* creates and sends a DHCP *Discover* message, waits to receive a reply, extracts the IP address, subnet mask, and default router address from the reply and stores them in *Netdata*, and returns the IP address. The code can be found in file *dhcp.c*:

```
/* dhcp.c - getlocalip */

#include <xinu.h>

/*------------------------------------------------------------------------
 * getlocalip - use DHCP to obtain an IP address
 *------------------------------------------------------------------------
 */
```

```
uint32  getlocalip(void)
{
        struct  dhcpmsg dmsg;           /* holds outgoing DHCP discover */
                                        /*      message                 */
        struct  dhcpmsg dmsg2;          /* holds incoming DHCP offer    */
                                        /* and outgoing request message */
        uint32  xid;                    /* xid used for the exchange    */
        int32   i;                      /* retry counter                */
        int32   len;                    /* length of data read          */
        char    *optptr;                /* pointer to options area      */
        char    *eop;                   /* address of end of packet     */
        int32   msgtype;                /* type of DCHP message         */
        uint32  addrmask;               /* address mask for network     */
        uint32  routeraddr;             /* default router address       */

        if (NetData.ipvalid) {          /* already have an IP address   */
                return NetData.ipaddr;
        }
        udp_register(0, UDP_DHCP_SPORT, UDP_DHCP_CPORT);
        memcpy(&xid, NetData.ethaddr, 4);
                                        /* use 4 bytes from MAC as XID  */

        /* handcraft a DHCP Discover message in dmsg */

        dmsg.dc_bop = 0x01;             /* Outgoing request             */
        dmsg.dc_htype = 0x01;           /* hardware type is Ethernet    */
        dmsg.dc_hlen = 0x06;            /* hardware address length      */
        dmsg.dc_hops = 0x00;            /* Hop count                    */
        dmsg.dc_xid = xid;              /* xid (unique ID)              */
        dmsg.dc_secs = 0x0000;          /* seconds                      */
        dmsg.dc_flags = 0x0000;         /* flags                        */
        dmsg.dc_cip = 0x00000000;       /* Client IP address            */
        dmsg.dc_yip = 0x00000000;       /* Your IP address              */
        dmsg.dc_sip = 0x00000000;       /* Server IP address            */
        dmsg.dc_gip = 0x00000000;       /* Gateway IP address           */
        memset(&dmsg.dc_chaddr,'\0',16);/* Client hardware address      */
        memcpy(&dmsg.dc_chaddr, NetData.ethaddr, ETH_ADDR_LEN);
        memset(&dmsg.dc_bootp,'\0',192);/* zero the bootp area          */
        dmsg.dc_cookie = 0x63825363;    /* Magic cookie for DHCP        */

        dmsg.dc_opt[0] = 0xff & 53;     /* DHCP message type option     */
        dmsg.dc_opt[1] = 0xff &  1;     /* option length                */
        dmsg.dc_opt[2] = 0xff &  1;     /* DHCP Dicover message         */
        dmsg.dc_opt[3] = 0xff &  0;     /* Options padding              */
```

```
dmsg.dc_opt[4] = 0xff & 55;     /* DHCP parameter request list  */
dmsg.dc_opt[5] = 0xff &  2;     /* option length                */
dmsg.dc_opt[6] = 0xff &  1;     /* request subnet mask          */
dmsg.dc_opt[7] = 0xff &  3;     /* request default router addr. */

dmsg.dc_opt[8] = 0xff &  0;     /* options padding              */
dmsg.dc_opt[9] = 0xff &  0;     /* options padding              */
dmsg.dc_opt[10]= 0xff &  0;     /* options padding              */
dmsg.dc_opt[11]= 0xff &  0;     /* options padding              */

len = (char *)&dmsg.dc_opt[11] - (char *)&dmsg + 1;

udp_send(IP_BCAST, UDP_DHCP_SPORT, IP_THIS, UDP_DHCP_CPORT,
                        (char *)&dmsg, len);

/* Read 3 incoming DHCP messages and check for an offer or      */
/*    wait for three timeout periods if no message arrives.     */

for (i=0; i<3; i++) {
        if ((len=udp_recv(0, UDP_DHCP_SPORT, UDP_DHCP_CPORT,
                        (char *)&dmsg2, sizeof(struct dhcpmsg),
                                        3000)) == TIMEOUT) {
                continue;
        }

        /* Check that incoming message is a valid response (ID  */
        /* matches our request)                                 */

        if ( (dmsg2.dc_xid != xid) ) {
                continue;
        }

        eop = (char *)&dmsg2 + len - 1;
        optptr = (char *)&dmsg2.dc_opt;
        msgtype = addrmask = routeraddr = 0;
        while (optptr < eop) {

            switch (*optptr) {
                case 53:        /* message type */
                        msgtype = 0xff & *(optptr+2);
                        break;

                case 1:         /* subnet mask */
                        memcpy(&addrmask, optptr+2, 4);
                        break;
```

```
                        case 3:         /* router address */
                                memcpy(&routeraddr, optptr+2, 4);
                                break;
                    }
                    optptr++;   /* move to length octet */
                    optptr += (0xff & *optptr) + 1;
                }

                if (msgtype == 0x02) {  /* offer - send request */
                        dmsg2.dc_opt[0] = 0xff & 53;
                        dmsg2.dc_opt[1] = 0xff &  1;
                        dmsg2.dc_opt[2] = 0xff &  3;
                        dmsg2.dc_bop = 0x01;
                        udp_send(IP_BCAST, UDP_DHCP_SPORT, IP_THIS,
                                UDP_DHCP_CPORT, (char *)&dmsg2,
                                        sizeof(struct dhcpmsg) - 4);

                } else if (dmsg2.dc_opt[2] != 0x05) {   /* if not an ack*/
                        continue;                       /* skip it      */
                }
                if (addrmask != 0) {
                        NetData.addrmask = addrmask;
                }
                if (routeraddr != 0) {
                        NetData.routeraddr = routeraddr;
                }
                NetData.ipaddr = dmsg2.dc_yip;
                NetData.ipvalid = TRUE;
                udp_release(0, UDP_DHCP_SPORT, UDP_DHCP_CPORT);
                return NetData.ipaddr;
        }
        kprintf("DHCP failed to get response\r\n");
        udp_release(0, UDP_DHCP_SPORT, UDP_DHCP_CPORT);
        return (uint32)SYSERR;
}
```

A DHCP server responds to the initial *Discover* message by sending the requested information. When it receives a reply, *getlocalip* examines the options area of the message. DHCP is unusual because the options carry information. In particular, the type of the DHCP message is stored in the options area as well as information that a computer system uses to initialize network parameters. Three options are crucial to our implementation: option 53 defines the *type* of a DHCP message, option 1 specifies a subnet mask used on the local network, and option 3 specifies the address of a default router.

If the options are present, *getlocalip* can extract the needed information from the reply, store the information for successive calls, and return the IP address to the caller.

The details of DHCP are beyond the scope of this text. However, it is sufficient to understand that the DHCP code uses the UDP interface exactly the same way that other applications use it. That is, before communication can begin, *getlocalip* must call *udp_register* to register the port that DHCP will use. Once the port has been registered, *getlocalip* can create a DHCP *Discover* message and call *udp_send* to broadcast the message. The DHCP *Discover* message causes a DHCP server to respond, and the system can obtain its IP address.

## 17.11 Perspective

The implementation of Internet protocols described in this chapter is indeed minimalistic. Many details have been omitted, and the code takes many shortcuts. For example, the structures used to define message formats combine multiple layers of the protocol stack and assume that the underlying network is always an Ethernet. Therefore, you should not view the code as typical of a protocol implementation nor should you assume that the same structure will suffice for a complete protocol stack.

Despite its limitations, the code does illustrate the importance of timed operations. In particular, the availability of a timed receive function simplifies the overall structure of the code and makes the operation much easier to understand. If the system did not provide a timed receive, multiple processes would be needed — one process would implement a timer function and another would handle responses.

## 17.12 Summary

Even small embedded systems use Internet protocols to communicate. As a consequence, most operating systems include software known as a protocol stack.

The chapter examines a minimal protocol stack that supports limited versions of IP, UDP, ICMP, ARP, and DHCP running over an Ethernet. The above protocols are closely interrelated. Both ICMP and UDP messages travel in an IP datagram; a DHCP message travels in a UDP packet.

To accommodate asynchronous packet arrivals, our protocol implementation uses a network input process, *netin*. The *netin* process repeatedly reads an Ethernet packet, validates header fields, and uses header information to determine how to process the packet. When an ARP packet arrives, *netin* calls *arp_in* to handle the packet, when a UDP packet arrives, *netin* calls *udp_in*, and when an ICMP packet arrives, *netin* calls *icmp_in*. For all other packets, *netin* ignores the packet. When receiving packets, our implementation allows a process to specify the maximum time to wait for a packet to arrive. The timeout mechanism can be used to implement retransmission: if a response does not arrive before the timeout occurs, a process can retransmit the request.

## EXERCISES

**17.1** Rewrite the code to eliminate the need for a separate ICMP output process. Hint: keep a queue of outgoing IP packets with each ARP table entry and arrange for the packets to be sent once an ARP reply arrives.

**17.2** The UDP functions each require a caller to specify endpoint information, such as IP addresses and protocol port numbers. Rewrite the code to change the paradigm: have *udp_register* return the index of an entry in the table, and have other functions, such as *udp_recv* take the index as an argument.

**17.3** Redesign the UDP protocol software to use timer processes instead of *recvtime*. How many processes are needed? Explain.

**17.4** Xinu uses a device paradigm for abstract devices as well as hardware devices. Rewrite the UDP code to use a device paradigm in which a process calls *open* on a UDP master device to specify protocol port and IP address information, and receives the descriptor of a pseudo-device that can be used for communication.

**17.5** Can the device paradigm described in the previous exercise handle all of ICMP? Does the answer change if the question is restricted to ICMP echo (i.e., *ping*)? Explain.

# Chapter Contents

# 18

# A Remote Disk Driver

*For my purpose holds ... To strive, to seek, to find, and not to yield.*

— Alfred, Lord Tennyson

## 18.1 Introduction

Earlier chapters explain I/O devices and the structure of a device driver. Chapter 16 describes how block-oriented devices use DMA, and shows an example Ethernet driver.

This chapter considers the design of a device driver for secondary storage devices known as *disks* or *hard drives*. The chapter focuses on basic data transfer operations. The next chapter describes how higher levels of the system use disk hardware to provide *files* and *directories*.

## 18.2 The Disk Abstraction

Disk hardware provides a basic abstraction in which a disk is a storage mechanism that has the following properties.

- *Nonvolatile*: data persists even if power is removed.
- *Block-oriented*: the interface provides the ability to read or write fixed-size blocks of data.
- *Multi-use*: a block can be read and written many times.
- *Random-access*: blocks can be accessed in any order.

Like the Ethernet hardware described in Chapter 16, disk hardware typically uses *Direct-Memory-Access* (*DMA*) to allow the disk to transfer an entire block before interrupting the CPU. Also like the Ethernet driver, a disk driver does not understand or examine the contents of data blocks. Instead, the driver merely treats the entire disk as an array of data blocks.

## 18.3 Operations A Disk Driver Supports

At the device driver level, a disk consists of fixed-size data blocks that can be accessed randomly using three basic operations:

- *Fetch*: Copy the contents of a specified block from the disk to a designated location in memory.
- *Store*: Copy the contents of memory to a specified block on the disk.
- *Seek*: Move to a specified block on the disk. The seek option is most important for electro-mechanical devices (i.e., a magnetic disk) because it can be used as an optimization that positions the disk head where it will be needed in the future. Thus, as solid state disks become widely used, seek may become less important.

The block size of a disk is derived from the size of a *sector* on magnetic disks. The industry has settled on a de facto standard block size of 512 bytes; throughout the chapter, we will assume 512-byte blocks.†

## 18.4 Block Transfer And High-Level I/O Functions

Because the hardware provides block transfer, it makes sense to design an interface that allows *read* and *write* to transfer an entire block. The question becomes how to include a block specification in the existing high-level I/O operations. We might use *seek*: require a programmer to call *seek* to move to a specific block before calling *read* or *write* to access data in the block. Unfortunately, requiring a user to call *seek* before each data transfer is clumsy and error prone. Therefore, to keep the interface simple, we will stretch the usual meaning of arguments to *read* and *write*: instead of interpreting the third argument as a buffer size, we will assume the buffer is large enough to hold a disk block, and use the third argument to specify a block number. For example, the call:

```
read ( DISK0, buff, 5 )
```

requests the driver to read block five from the disk into memory starting at location *buff*.

†Although modern disks often use an underlying block size of 4K bytes, the hardware presents an interface that uses 512-byte blocks.

Our driver will supply basic operations: *read*, which copies a single block from the disk into memory; *write*, which copies data from memory onto a specified disk block. In addition, our driver will supply *control* functions that can be used to erase a disk (i.e., destroy all saved data) and to synchronize *write* requests (i.e., insure that all cached data is written to the disk).

## 18.5 A Remote Disk Paradigm

Because the E2100L hardware does not include a local disk, we will examine software that follows a *remote disk* paradigm. Our remote disk system will provide the same abstraction as a local disk by allowing processes to read and write disk blocks. Instead of using disk hardware, however, our system will send requests over a network to a remote disk server that runs on another computer.

Driver software for the remote disk will be organized in the same general way as a driver for a traditional disk: driver functions are partitioned into an upper half and a lower half that communicate through a shared data structure. Instead of using DMA hardware to implement the lower half, however, our remote disk driver will use a high-priority process that communicates over a network with a server by sending requests and receiving responses. Figure 18.1 illustrates the organization.

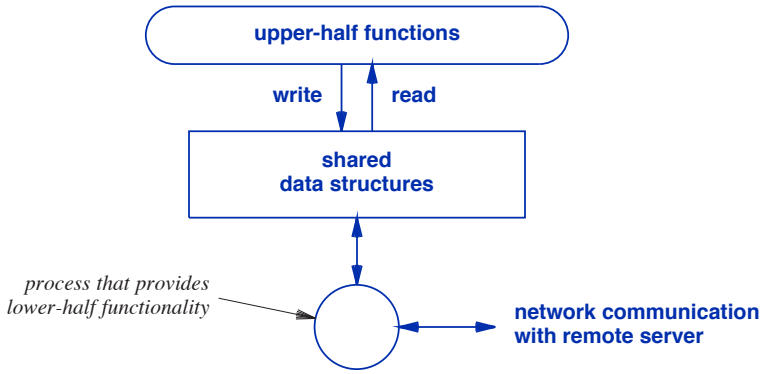



**Figure 18.1** Organization of a remote disk driver.

Like a traditional disk driver, the shared data structure in our example contains two key items:

- A cache of recently accessed blocks
- A list of pending requests

*A cache of recently accessed blocks*. Because a disk operates much slower than a processor, a disk driver must be optimized to avoid making unnecessary transfers. The primary optimization technique consists of a *cache* kept in the shared data area. The cache stores recently accessed blocks, which allows subsequent operations on the same block to be performed on the copy in the cache. For example, if a user edits a document, stores a copy on disk, and then returns later to edit the same document, it is likely that the disk blocks containing the document will be in the cache, meaning that the document editing application can start much faster than if blocks had to be read from disk.

*A list of pending requests*. Like a traditional driver, our remote disk system allows multiple processes to access a disk, and implements synchronous *read* operations and asynchronous *write* operations. That is, when it reads a block, a process must wait until the data has been fetched. When it writes a block, however, a process does not block — the driver places a copy of the outgoing disk block on a request list and allows the process to continue execution. The lower-half process, which handles communication with the remote server, continually takes the next item off the request queue and performs the specified operation. Thus, at some time in the future, data will be written to the disk.

## 18.6 Semantics Of Disk Operations

Although it can employ optimizations such as caching and delayed execution, a disk driver must always preserve the appearance of a synchronous interface. That is, the driver must always return the data that was written most recently.

If we envision the *read* and *write* operations on a given block, they form a sequence:

$$op_1,\ op_2,\ op_3, \ldots\ op_n$$

If $op_t$ is a *read* operation for block *i*, the driver must deliver the data that was written to the block in $op_k$, where *k* is the highest index of a *write* operation less than *t* (i.e., all the operations between $op_k$ and $op_t$ are *read* operations). To complete the definition, we assume an implicit *write* occurs at time zero before the system starts. Thus, if a system attempts to *read* a block before any calls to *write* the block, the driver returns whatever data was on the disk when the system booted.

We use the term *last-write semantics* to capture the concept, and insist that:

> *A disk driver can use techniques such as caching to optimize performance provided the driver guarantees last-write semantics.*

The example driver uses queuing to enforce last-write semantics: the request list is a FIFO queue. That is:

*Items are inserted at the tail of the request queue; the lower-half process continually selects and performs the item at the head of the queue.*

Because items are always inserted at the tail, the driver handles requests in the same order they were made. Thus, if process A *reads* block five and process B *writes* block five later, the two requests will appear in the correct order in the queue. The *read* request will be serviced first, followed by the *write* request.

We will see that the queue discipline is extended to the cache: driver functions always start at the head when searching the cache. Our code relies on the discipline to insure that a process receives data according to last-write semantics.

## 18.7 Definition Of Driver Data Structures

File *rdisksys.h* defines the constants and data structures used by the remote disk system. The file defines the format of a disk buffer. Each buffer includes a header that specifies the number of the disk block stored in the buffer and fields that are used to link the buffer onto the request list, cache, or free list. In addition, the file defines the contents of the device control block and the format of messages sent to the remote server.

```
/* rdisksys.h - definitions for remote disk system pseudo-devices */

#ifndef Nrds
#define Nrds            1
#endif

/* Remote disk block size */

#define RD_BLKSIZ       512

/* Global data for the remote disk server */

#ifndef RD_SERVER_IP
#define RD_SERVER_IP    "255.255.255.255"
#endif

#ifndef RD_SERVER_PORT
#define RD_SERVER_PORT  33124
#endif

#ifndef RD_LOC_PORT
#define RD_LOC_PORT     33124           /* base port number - minor dev */
                                        /*   number is added to insure  */
                                        /*   that each device is unique */
#endif

/* Control block for remote disk device */

#define RD_IDLEN        64              /* Size of a remote disk ID     */
#define RD_BUFFS        64              /* Number of disk buffers       */
#define RD_STACK        8192            /* Stack size for comm. process */
#define RD_PRIO         200             /* Priorty of comm. process     */

/* Constants for state of the device */

#define RD_FREE          0              /* device is available          */
#define RD_OPEN          1              /* device is open (in use)      */
#define RD_PEND          2              /* open is pending              */

/* Operations for request queue */

#define RD_OP_READ      1               /* Read operation on req. list  */
#define RD_OP_WRITE     2               /* Write operation on req. list */
#define RD_OP_SYNC      3               /* Sync operation on req. list  */
```

```
/* Status values for a buffer */

#define RD_VALID        0               /* Buffer contains valid data   */
#define RD_INVALID      1               /* Buffer does not contain data */

/* Definition of a buffer with a header that allows the same node to be */
/*  used as a request on the request queue, an item in the cache, or a  */
/*  node on the free list of buffers                                    */

struct  rdbuff  {                       /* Request list node            */
        struct  rdbuff  *rd_next;       /* ptr to next node on a list   */
        struct  rdbuff  *rd_prev;       /* ptr to prev node on a list   */
        int32   rd_op;                  /* operation - read/write/sync  */
        int32   rd_refcnt;              /* reference count of processes */
                                        /*   reading the block          */
        uint32  rd_blknum;              /* block number of this block   */
        int32   rd_status;              /* is buffer currently valid?   */
        pid32   rd_pid;                 /* process that initiated a     */
                                        /*   read request for the block */
        char    rd_block[RD_BLKSIZ];    /* space to hold one disk block */
};

struct  rdscblk {
        int32   rd_state;               /* state of device              */
        char    rd_id[RD_IDLEN];        /* Disk ID currently being used */
        int32   rd_seq;                 /* next sequence number to use  */

        /* Request queue head and tail */

        struct  rdbuff  *rd_rhnext;     /* head of request queue: next  */
        struct  rdbuff  *rd_rhprev;     /*  and previous                */
        struct  rdbuff  *rd_rtnext;     /* tail of request queue: next  */
        struct  rdbuff  *rd_rtprev;     /*  (null) and previous         */

        /* Cache head and tail */

        struct  rdbuff  *rd_chnext;     /* head of cache: next and      */
        struct  rdbuff  *rd_chprev;     /*  previous                    */
        struct  rdbuff  *rd_ctnext;     /* tail of cache: next (null)   */
        struct  rdbuff  *rd_ctprev;     /*  and previous                */

        /* Free list head (singly-linked) */

        struct  rdbuff  *rd_free;       /* ptr to free list             */

        pid32   rd_comproc;             /* process ID of comm. process  */
```

```
        sid32   rd_availsem;            /* semaphore ID for avail buffs */
        sid32   rd_reqsem;              /* semaphore ID for requests    */
        uint32  rd_ser_ip;              /* server IP address            */
        uint16  rd_ser_port;            /* server UDP port              */
        uint16  rd_loc_port;            /* local (client) UPD port      */
        bool8   rd_registered;          /* has UDP port been registered?*/
};


extern  struct  rdscblk rdstab[];       /* remote disk control block    */

/* Definitions of parameters used during server access */

#define RD_RETRIES      3               /* times to retry sending a msg */
#define RD_TIMEOUT      2000            /* wait two seconds for reply   */

/* Control functions for a remote file pseudo device */

#define RDS_CTL_DEL     1               /* Delete (erase) an entire disk*/
#define RDS_CTL_SYNC    2               /* Write all pending blocks     */

/************************************************************************/
/*      Definition of messages exchanged with the remote disk server    */
/************************************************************************/
/* Values for the type field in messages */

#define RD_MSG_RESPONSE 0x0100          /* Bit that indicates response  */

#define RD_MSG_RREQ     0x0010          /* Read request and response    */
#define RD_MSG_RRES     (RD_MSG_RREQ | RD_MSG_RESPONSE)

#define RD_MSG_WREQ     0x0020          /* Write request and response   */
#define RD_MSG_WRES     (RD_MSG_WREQ | RD_MSG_RESPONSE)

#define RD_MSG_OREQ     0x0030          /* Open request and response    */
#define RD_MSG_ORES     (RD_MSG_OREQ | RD_MSG_RESPONSE)

#define RD_MSG_CREQ     0x0040          /* Close request and response   */
#define RD_MSG_CRES     (RD_MSG_CREQ | RD_MSG_RESPONSE)

#define RD_MSG_DREQ     0x0050          /* Delete request and response  */
#define RD_MSG_DRES     (RD_MSG_DREQ | RD_MSG_RESPONSE)

#define RD_MIN_REQ      RD_MSG_RREQ     /* Minimum request type         */
#define RD_MAX_REQ      RD_MSG_DREQ     /* Maximum request type         */
```

```
/* Message header fields present in each message */

#define RD_MSG_HDR                      /* Common message fields        */\
        uint16  rd_type;                /* message type                 */\
        uint16  rd_status;              /* 0 in req, status in response */\
        uint32  rd_seq;                 /* message sequence number      */\
        char    rd_id[RD_IDLEN];        /* null-terminated disk ID      */

/************************************************************************/
/*                              Header                                  */
/************************************************************************/
/* The standard header present in all messages with no extra fields */
#pragma pack(2)
struct  rd_msg_hdr {                    /* header fields present in each*/
        RD_MSG_HDR                      /*   remote file system message */
};
#pragma pack()

/************************************************************************/
/*                              Read                                    */
/************************************************************************/
#pragma pack(2)
struct  rd_msg_rreq     {               /* remote file read request     */
        RD_MSG_HDR                      /* header fields                */
        uint32  rd_blk;                 /* block number to read         */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_rres     {               /* remote file read reply       */
        RD_MSG_HDR                      /* header fields                */
        uint32  rd_blk;                 /* block number that was read   */
        char    rd_data[RD_BLKSIZ];     /* array containing one block   */
};
#pragma pack()

/************************************************************************/
/*                              Write                                   */
/************************************************************************/
#pragma pack(2)
struct  rd_msg_wreq     {               /* remote file write request    */
        RD_MSG_HDR                      /* header fields                */
        uint32  rd_blk;                 /* block number to write        */
        char    rd_data[RD_BLKSIZ];     /* array containing one block   */
};
#pragma pack()
```

```
#pragma pack(2)
struct  rd_msg_wres     {               /* remote file write response   */
        RD_MSG_HDR                      /* header fields                */
        uint32  rd_blk;                 /* block number that was written*/
};
#pragma pack()

/*************************************************************************/
/*                              Open                                     */
/*************************************************************************/
#pragma pack(2)
struct  rd_msg_oreq     {               /* remote file open request     */
        RD_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_ores     {               /* remote file open response    */
        RD_MSG_HDR                      /* header fields                */
};
#pragma pack()

/*************************************************************************/
/*                              Close                                    */
/*************************************************************************/
#pragma pack(2)
struct  rd_msg_creq     {               /* remote file close request    */
        RD_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_cres     {               /* remote file close response   */
        RD_MSG_HDR                      /* header fields                */
};
#pragma pack()

/*************************************************************************/
/*                              Delete                                   */
/*************************************************************************/
#pragma pack(2)
struct  rd_msg_dreq     {               /* remote file delete request   */
        RD_MSG_HDR                      /* header fields                */
};
```

```
#pragma pack()

#pragma pack(2)
struct  rd_msg_dres     {               /* remote file delete response  */
        RD_MSG_HDR                      /* header fields                */
};
#pragma pack()
```

## 18.8 Driver Initialization (rdsInit)

Although initialization is completed after the other pieces of the driver have been designed, we have chosen to examine the initialization function now because it will help us understand the shared data structures. File *rdsInit.c* contains the driver initialization code:

```
/* rdsInit.c  -  rdsInit */

#include <xinu.h>

struct  rdscblk rdstab[Nrds];

/*------------------------------------------------------------------------
 *  rdsInit - initialize the remote disk system device
 *------------------------------------------------------------------------
 */
devcall rdsInit (
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  rdscblk *rdptr;         /* ptr to device contol block   */
        struct  rdbuff  *bptr;          /* ptr to buffer in memory      */
                                        /*    used to form linked list  */
        struct  rdbuff  *pptr;          /* ptr to previous buff on list */
        struct  rdbuff  *buffend;       /* last address in buffer memory*/
        uint32  size;                   /* total size of memory needed  */
                                        /*    buffers                   */

        /* Obtain address of control block */

        rdptr = &rdstab[devptr->dvminor];

        /* Set control block to unused */
```

```
	rdptr->rd_state = RD_FREE;
	rdptr->rd_id[0] = NULLCH;

	/* Set initial message sequence number */

	rdptr->rd_seq = 1;

	/* Initialize request queue and cache to empty */

	rdptr->rd_rhnext = (struct rdbuff *) &rdptr->rd_rtnext;
	rdptr->rd_rhprev = (struct rdbuff *)NULL;

	rdptr->rd_rtnext = (struct rdbuff *)NULL;
	rdptr->rd_rtprev = (struct rdbuff *) &rdptr->rd_rhnext;


	rdptr->rd_chnext = (struct rdbuff *) &rdptr->rd_ctnext;
	rdptr->rd_chprev = (struct rdbuff *)NULL;

	rdptr->rd_ctnext = (struct rdbuff *)NULL;
	rdptr->rd_ctprev = (struct rdbuff *) &rdptr->rd_chnext;

	/* Allocate memory for a set of buffers (actually request       */
	/*    blocks and link them to form the initial free list        */

	size = sizeof(struct rdbuff) * RD_BUFFS;

	bptr = (struct rdbuff *)getmem(size);
	rdptr->rd_free = bptr;

	if ((int32)bptr == SYSERR) {
		panic("Cannot allocate memory for remote disk buffers");
	}

	buffend = (struct rdbuff *) ((char *)bptr + size);
	while (bptr < buffend) {        /* walk through memory */
		pptr = bptr;
		bptr = (struct rdbuff *)
				(sizeof(struct rdbuff)+ (char *)bptr);
		pptr->rd_status = RD_INVALID;   /* buffer is empty      */
		pptr->rd_next = bptr;           /* point to next buffer */
	}
	pptr->rd_next = (struct rdbuff *) NULL; /* last buffer on list  */

	/* Create the request list and available buffer semaphores */
```

```
        rdptr->rd_availsem = semcreate(RD_BUFFS);
        rdptr->rd_reqsem   = semcreate(0);

        /* Set the server IP address, server port, and local port */

        if ( dot2ip(RD_SERVER_IP, &rdptr->rd_ser_ip) == SYSERR ) {
                panic("invalid IP address for remote disk server");
        }

        /* Set the port numbers */

        rdptr->rd_ser_port = RD_SERVER_PORT;
        rdptr->rd_loc_port = RD_LOC_PORT + devptr->dvminor;

        /* Specify that the server port is not yet registered */

        rdptr->rd_registered = FALSE;

        /* Create a communication process */

        rdptr->rd_comproc = create(rdsprocess, RD_STACK, RD_PRIO,
                                                "rdsproc", 1, rdptr);

        if (rdptr->rd_comproc == SYSERR) {
                panic("Cannot create remote disk process");
        }
        resume(rdptr->rd_comproc);

        return OK;
}
```

In addition to initializing data structures, *rdsInit* performs three important tasks. It allocates a set of disk buffers and links them onto the free list, it creates the high-priority process that communicates with the server, and it creates two semaphores that control processing. One semaphore, *rd_reqsem*, guards the request list. The semaphore starts with count zero, and is signaled each time a new request is added to the request queue. The communication process waits on *rd_reqsem* before extracting an item from the list, which means the process will block if the list is empty.

Another semaphore, *rd_availsem*, counts the number of buffers that are available for use (i.e., free or in the cache). Initially, *RD_BUFFS* buffers are on the free list and *rd_availsem* has a count equal to *RD_BUFFS*. When a buffer is needed, a caller waits on the semaphore. The cache poses a special case because not all buffers in the cache are available. Instead, some buffers must remain in the cache because processes are waiting to extract data. We will see how the cache and the semaphore are used later.

## 18.9 The Upper–Half Open Function (rdsOpen)

The remote disk server allows multiple clients to access the server simultaneously. Each client supplies a unique identification string which allows the server to distinguish among clients. Instead of using a hardware value (e.g., an Ethernet address) as the unique string, the example code allows a user to specify the ID string by calling *open* on the disk device. The chief advantage of separating the ID from the hardware is portability — the remote disk ID can be bound to the operating system image, which means that moving the image from one physical computer to another does not change the disk that the system is using.

When a process calls *open* for a remote disk device, the second argument is interpreted as an ID string. The string is copied into the device control block, and the same ID is used as long as the device remains open. It is possible to close the remote disk device and reopen it with a new ID (i.e., connect to a different remote disk). However, most systems are expected to open a remote disk device once and never close it. File *rdsOpen.c* contains the code:

```
/* rdsOpen.c  -  rdsOpen */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsOpen - open a remote disk device and specify an ID to use
 *------------------------------------------------------------------------
 */

devcall rdsOpen (
         struct dentry  *devptr,        /* entry in device switch table */
         char   *diskid,                /* disk ID to use               */
         char   *mode                   /* unused for a remote disk     */
        )
{
        struct  rdscblk *rdptr;         /* ptr to control block entry   */
        struct  rd_msg_oreq msg;        /* message to be sent           */
        struct  rd_msg_ores resp;       /* buffer to hold response      */
        int32   retval;                 /* return value from rdscomm    */
        int32   len;                    /* counts chars in diskid       */
        char    *idto;                  /* ptr to ID string copy        */
        char    *idfrom;                /* pointer into ID string       */

        rdptr = &rdstab[devptr->dvminor];

        /* Reject if device is already open */
```

```
        if (rdptr->rd_state != RD_FREE) {
                return SYSERR;
        }
        rdptr->rd_state = RD_PEND;

        /* Copy disk ID into free table slot */

        idto = rdptr->rd_id;
        idfrom = diskid;
        len = 0;
        while ( (*idto++ = *idfrom++) != NULLCH) {
                len++;
                if (len >= RD_IDLEN) {  /* ID string is too long */
                        return SYSERR;
                }
        }

        /* Verify that name is non-null */

        if (len == 0) {
                return SYSERR;
        }

        /* Hand-craft an open request message to be sent to the server */

        msg.rd_type = htons(RD_MSG_OREQ);/* Request an open             */
        msg.rd_status = htons(0);
        msg.rd_seq = 0;                 /* rdscomm fills in an entry    */
        idto = msg.rd_id;
        memset(idto, NULLCH, RD_IDLEN);/* initialize ID to zero bytes   */

        idfrom = diskid;
        while ( (*idto++ = *idfrom++) != NULLCH ) { /* copy ID to req.  */
                ;
        }

        /* Send message and receive response */

        retval = rdscomm((struct rd_msg_hdr *)&msg,
                                        sizeof(struct rd_msg_oreq),
                        (struct rd_msg_hdr *)&resp,
                                        sizeof(struct rd_msg_ores),
                                rdptr );

        /* Check response */
```

```
        if (retval == SYSERR) {
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file open\n\r");
                return SYSERR;
        } else if (ntohs(resp.rd_status) != 0) {
                return SYSERR;
        }

        /* Change state of device to indicate currently open */

        rdptr->rd_state = RD_OPEN;

        /* Return device descriptor */

        return devptr->dvnum;
}
```

## 18.10 The Remote Communication Function (rdscomm)

As one of the steps in opening the local remote disk device, *rdsOpen* exchanges a message with the remote server. It places an *open request* message in local variable *msg*, and calls *rdscomm* to forward the message to the server. *Rdscomm* takes arguments that specify an outgoing message, a buffer for a reply, and the length of each. It sends the outgoing message to the server, and awaits a reply. If the reply is valid, *rdscomm* returns the length of the reply to the caller; otherwise, it returns *SYSERR* to indicate that an error occurred, or *TIMEOUT* to indicate that no response was received. File *rdscomm.c* contains the code:

```
/* rdscomm.c  -  rdscomm */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdscomm - handle communication with a remote disk server (send a
 *              request and receive a reply, including sequencing and
 *              retries)
 *------------------------------------------------------------------------
 */
status  rdscomm (
          struct rd_msg_hdr *msg,       /* message to send              */
          int32             mlen,       /* message length               */
```

```
          struct rd_msg_hdr *reply,     /* buffer for reply             */
          int32             rlen,       /* size of reply buffer         */
          struct rdscblk    *rdptr      /* ptr to device control block  */
        )
{
        int32   i;                      /* counts retries               */
        int32   retval;                 /* return value                 */
        int32   seq;                    /* sequence for this exchange   */
        uint32  localip;                /* local IP address             */
        int16   rtype;                  /* reply type in host byte order*/
        bool8   xmit;                   /* Should we transmit again?    */

        /* For the first time after reboot, register the server port */

        if ( ! rdptr->rd_registered ) {
                retval = udp_register(0, rdptr->rd_ser_port,
                                rdptr->rd_loc_port);
                rdptr->rd_registered = TRUE;
        }

        if ( (int32)(localip = getlocalip()) == SYSERR ) {
                return SYSERR;
        }

        /* Assign message next sequence number */

        seq = rdptr->rd_seq++;
        msg->rd_seq = htonl(seq);

        /* Repeat RD_RETRIES times: send message and receive reply */

        xmit = TRUE;
        for (i=0; i<RD_RETRIES; i++) {
            if (xmit) {

                /* Send a copy of the message */

                retval = udp_send(rdptr->rd_ser_ip, rdptr->rd_ser_port,
                        localip, rdptr->rd_loc_port, (char *)msg, mlen);
                if (retval == SYSERR) {
                        kprintf("Cannot send to remote disk server\n\r");
                        return SYSERR;
                }
            } else {
                xmit = TRUE;
            }
```

```
            /* Receive a reply */

            retval = udp_recv(0, rdptr->rd_ser_port,
                rdptr->rd_loc_port, (char *)reply, rlen,
                RD_TIMEOUT);

            if (retval == TIMEOUT) {
                continue;
            } else if (retval == SYSERR) {
                kprintf("Error reading remote disk reply\n\r");
                return SYSERR;
            }

            /* Verify that sequence in reply matches request */


            if (ntohl(reply->rd_seq) < seq) {
                xmit = FALSE;
            } else if (ntohl(reply->rd_seq) != seq) {
                        continue;
            }

            /* Verify the type in the reply matches the request */

            rtype = ntohs(reply->rd_type);
            if (rtype != ( ntohs(msg->rd_type) | RD_MSG_RESPONSE) ) {
                continue;
            }

            /* Check the status */

            if (ntohs(reply->rd_status) != 0) {
                return SYSERR;
            }

            return OK;
        }

        /* Retries exhausted without success */

        kprintf("Timeout on exchange with remote disk server\n\r");
        return TIMEOUT;
}
```

*Rdscomm* obtains the local IP address (for use with UDP), assigns the next sequence number to the message, and enters a loop that iterates *RD_RETRIES* times. On each iteration, *rdscomm* calls *udp_send* to transmit a copy of the message to the server, and calls *udp_recv* to receive a reply. If a reply arrives, *rdscomm* verifies that the *type* of the reply matches the type of the request, that the sequence number of the reply matches the sequence number of the request that was sent, and that the status value indicates success (i.e., is zero). If the reply is valid, *rdscomm* returns *OK* to the caller; otherwise, it returns an error indication.

## 18.11 The Upper–Half Write Function (rdsWrite)

Because the remote disk system provides asynchronous *write* operations, the upper-half *write* function is easiest to understand. File *rdsWrite.c* contains the code:

```
/* rdsWrite.c  -  rdsWrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsWrite - Write a block to a remote disk
 *------------------------------------------------------------------------
 */
devcall rdsWrite (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer that holds a disk blk */
          int32 blk                     /* block number to write        */
        )
{
        struct  rdscblk *rdptr;         /* pointer to control block     */
        struct  rdbuff  *bptr;          /* ptr to buffer on a list      */
        struct  rdbuff  *pptr;          /* ptr to previous buff on list */
        struct  rdbuff  *nptr;          /* ptr to next buffer on list   */
        bool8   found;                  /* was buff found during search?*/

        /* If device not currently in use, report an error */

        rdptr = &rdstab[devptr->dvminor];
        if (rdptr->rd_state != RD_OPEN) {
                return SYSERR;
        }

        /* If request queue already contains a write request */
        /*    for the block, replace the contents            */
```

```
bptr = rdptr->rd_rhnext;
while (bptr != (struct rdbuff *)&rdptr->rd_rtnext) {
        if ( (bptr->rd_blknum == blk) &&
                (bptr->rd_op == RD_OP_WRITE) ) {
                memcpy(bptr->rd_block, buff, RD_BLKSIZ);
                return OK;
        }
        bptr = bptr->rd_next;
}


/* Search cache for cached copy of block */

bptr = rdptr->rd_chnext;
found = FALSE;
while (bptr != (struct rdbuff *)&rdptr->rd_ctnext) {
        if (bptr->rd_blknum == blk) {
                if (bptr->rd_refcnt <= 0) {
                        pptr = bptr->rd_prev;
                        nptr = bptr->rd_next;

                        /* Unlink node from cache list and reset*/
                        /*  the available semaphore accordingly */

                        pptr->rd_next = bptr->rd_next;
                        nptr->rd_prev = bptr->rd_prev;
                        semreset(rdptr->rd_availsem,
                                semcount(rdptr->rd_availsem) - 1);
                        found = TRUE;
                }
                break;
        }
        bptr = bptr->rd_next;
}

if ( !found ) {
        bptr = rdsbufalloc(rdptr);
}

/* Create a write request */

memcpy(bptr->rd_block, buff, RD_BLKSIZ);
bptr->rd_op = RD_OP_WRITE;
bptr->rd_refcnt = 0;
```

```
        bptr->rd_blknum = blk;
        bptr->rd_status = RD_VALID;
        bptr->rd_pid = getpid();

        /* Insert new request into list just before tail */

        pptr = rdptr->rd_rtprev;
        rdptr->rd_rtprev = bptr;
        bptr->rd_next = pptr->rd_next;
        bptr->rd_prev = pptr;
        pptr->rd_next = bptr;

        /* Signal semaphore to start communication process */

        signal(rdptr->rd_reqsem);
        return OK;
}
```

The code first considers the case where the request queue already contains a pending *write* request for the same block. Note: there can only be one *write* request for a given block on the request queue at a time, which means the queue can be searched in any order. The code searches the queue from the head toward the tail. If it finds a *write* request for the block, *rdsWrite* replaces the contents of the request with the new data, and returns.

After searching the request queue, *rdsWrite* checks the cache. If the specified block is in the cache, the cached copy must be invalidated. The code searches the cache sequentially. If it finds a match, *rdsWrite* removes the buffer from the cache. Instead of moving the buffer to the free list, *rdsWrite* uses the buffer to form a request. If no match is found, *rdsWrite* calls *rdsbufalloc* to allocate a new buffer for the request.

The final section of *rdsWrite* creates a *write request* and inserts it at the tail of the request queue. To help with debugging, the code fills in all fields of the request, even if they are not needed. For example, the process ID field is set, but is not used.

## 18.12 The Upper–Half Read Function (rdsRead)

The second major upper-half function corresponds to the *read* operation. Reading is more complex than writing, because input is synchronous: a process that attempts to read from the disk must wait until the data is available. Synchronization of a waiting process uses *send* and *receive*. A node in the request queue contains a process ID field. When a process calls *read*, the driver code creates a *read* request that includes the caller's process ID. It then inserts the request on the request queue, and calls *receive* to wait for a response. When the request reaches the head of the queue, the remote disk communication process sends a message to the server and receives a response that con-

tains the specified block. The communication process copies the block into the buffer that contains the original request, moves the buffer to the cache, and uses *send* to send a message to the waiting process with the buffer address. The waiting process receives the message, extracts a copy of the data, and returns to the function that called *read*.

As described, the above scheme is insufficient because buffers are used dynamically. To understand the problem, imagine that a low-priority process is blocked waiting to read block 5. Eventually, the communication process obtains block 5 from the server, stores block 5 in the cache, and sends a message to the waiting process. However, assume that while the request is on the request queue, higher-priority application processes begin to execute, meaning that the low-priority process will not run. Unfortunately, if the high-priority processes continue to use disk buffers, the buffer holding block 5 will be allocated.

The problem is exacerbated because the remote disk system permits concurrent access: while one process is waiting to read a block, another process can attempt to read the same block. Thus, when the communication process finally retrieves a copy of the block from the server, multiple processes may need to be informed.

The example code uses a *reference count* technique to handle multiple requests for a block: the header with each buffer contains an integer that counts the number of processes reading the block. When a process finishes making a copy, the process decrements the reference count. The code in file *rdsRead.c* shows how a process creates a request, enqueues it at the tail of the request list, waits for the request to be filled, and copies data from the request to each caller's buffer; later in the chapter, we will see how the reference count is managed.

```
/* rdsRead.c  -  rdsRead */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsRead - Read a block from a remote disk
 *------------------------------------------------------------------------
 */
devcall rdsRead (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer to hold disk block    */
          int32 blk                     /* block number of block to read*/
        )
{
        struct  rdscblk *rdptr;         /* pointer to control block     */
        struct  rdbuff  *bptr;          /* ptr to buffer possibly on    */
                                        /*  the request list            */
        struct  rdbuff  *nptr;          /* ptr to "next" node on a      */
                                        /*  list                        */
        struct  rdbuff  *pptr;          /* ptr to "previous" node on    */
```

```
                                        /*  a list                          */
struct  rdbuff  *cptr;                  /* ptr used to walk the cache       */

/* If device not currently in use, report an error */

rdptr = &rdstab[devptr->dvminor];
if (rdptr->rd_state != RD_OPEN) {
        return SYSERR;
}

/* Search the cache for specified block */

bptr = rdptr->rd_chnext;
while (bptr != (struct rdbuff *)&rdptr->rd_ctnext) {
        if (bptr->rd_blknum == blk) {
                if (bptr->rd_status == RD_INVALID) {
                        return SYSERR;
                }
                memcpy(buff, bptr->rd_block, RD_BLKSIZ);
                return OK;
        }
        bptr = bptr->rd_next;
}

/* Search the request list for most recent occurrence of block */

bptr = rdptr->rd_rtprev;        /* start at tail of list */

while (bptr != (struct rdbuff *)&rdptr->rd_rhnext) {
    if (bptr->rd_blknum == blk)  {

        /* If most recent request for block is write, copy data */

        if (bptr->rd_op == RD_OP_WRITE) {
                memcpy(buff, bptr->rd_block, RD_BLKSIZ);
                return OK;
        }
        break;
    }
    bptr = bptr->rd_prev;
}

/* Allocate a buffer and add read request to tail of req. queue */

bptr = rdsbufalloc(rdptr);
```

```
bptr->rd_op = RD_OP_READ;
bptr->rd_refcnt = 1;
bptr->rd_blknum = blk;
bptr->rd_status = RD_INVALID;
bptr->rd_pid = getpid();

/* Insert new request into list just before tail */

pptr = rdptr->rd_rtprev;
rdptr->rd_rtprev = bptr;
bptr->rd_next = pptr->rd_next;
bptr->rd_prev = pptr;
pptr->rd_next = bptr;

/* Prepare to receive message when read completes */

recvclr();

/* Signal semaphore to start communication process */

signal(rdptr->rd_reqsem);

/* Block to wait for message */

bptr = (struct rdbuff *)receive();
if (bptr == (struct rdbuff *)SYSERR) {
        return SYSERR;
}
memcpy(buff, bptr->rd_block, RD_BLKSIZ);
bptr->rd_refcnt--;
if (bptr->rd_refcnt <= 0) {

        /* Look for previous item in cache with the same block  */
        /*    number to see if this item was only being kept    */
        /*    until pending read completed                      */

        cptr = rdptr->rd_chnext;
        while (cptr != bptr) {
                if (cptr->rd_blknum == blk) {

                        /* Unlink from cache */

                        pptr = bptr->rd_prev;
                        nptr = bptr->rd_next;
                        pptr->rd_next = nptr;
```

```
                                nptr->rd_prev = pptr;

                                /* Add to the free list */

                                bptr->rd_next = rdptr->rd_free;
                                rdptr->rd_free = bptr;
                        }
                }
        }
        return OK;
}
```

*RdsRead* begins by handling two special cases. First, if the requested block is found in the cache, *rdsRead* extracts a copy of the data and returns. Second, if the request list contains a request to *write* the specified block, *rdsRead* extracts a copy of the data from the buffer and returns. Finally, *rdsRead* creates a read request, enqueues the request at the tail of the request queue, and waits for a message from the communication process as described above.

The code handles one more detail: the case where the reference count reaches zero and a subsequent *read* for the same block has placed a more recent buffer in the cache. If this happens, the more recent version will be used for subsequent *read*s. Therefore, *rdsRead* must extract the old buffer from the cache and move it to the free list.

## 18.13 Flushing Pending Requests

Because *write* does not wait for data transfer, the driver does not inform a process when a *write* operation completes. However, it may be important for the software to know when data is safely stored. For example, an operating system usually insures that *write* operations are completed before shutdown.

To allow a process to guarantee that all disk transfers have occurred, the driver includes a primitive that will block the calling process until existing requests have been performed. Because "synchronizing" the disk is not a data transfer operation, we use the high-level operation *control*. To flush pending requests, a process calls:

control ( disk_device, RD_SYNC )

The driver suspends the calling process until existing requests have been satisfied on the specified device. Once pending operations complete, the call returns.

## 18.14 The Upper–Half Control Function (rdsControl)

As discussed above, the example driver offers two *control* functions: one to erase an entire disk and one to synchronize data to the disk (i.e., forcing all *write* operations to complete). File *rdsControl.c* contains the code:

```
/* rdsControl.c  -  rdsControl */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsControl - Provide control functions for the remote disk
 *------------------------------------------------------------------------
 */
devcall rdsControl (
         struct dentry  *devptr,        /* entry in device switch table */
         int32  func,                   /* a control function           */
         int32  arg1,                   /* argument #1                  */
         int32  arg2                    /* argument #2                  */
        )
{
        struct  rdscblk *rdptr;         /* pointer to control block     */
        struct  rdbuff  *bptr;          /* ptr to buffer that will be   */
                                        /*  placed on the req. queue    */
        struct  rdbuff  *pptr;          /* ptr to "previous" node on    */
                                        /*  a list                      */
        struct  rd_msg_dreq msg;        /* buffer for delete request    */
        struct  rd_msg_dres resp;       /* buffer for delete response   */
        char    *to, *from;             /* used during name copy        */
        int32   retval;                 /* return value                 */

        /* Verify that device is currently open */

        rdptr = &rdstab[devptr->dvminor];
        if (rdptr->rd_state != RD_OPEN) {
                return SYSERR;
        }

        switch (func) {

        /* Synchronize writes */

        case RDS_CTL_SYNC:
```

```
                /* Allocate a buffer to use for the request list */

                bptr = rdsbufalloc(rdptr);
                if (bptr == (struct rdbuff *)SYSERR) {
                        return SYSERR;
                }

                /* Form a sync request */

                bptr->rd_op = RD_OP_SYNC;
                bptr->rd_refcnt = 1;
                bptr->rd_blknum = 0;            /* unused */
                bptr->rd_status = RD_INVALID;
                bptr->rd_pid = getpid();

                /* Insert new request into list just before tail */

                pptr = rdptr->rd_rtprev;
                rdptr->rd_rtprev = bptr;
                bptr->rd_next = pptr->rd_next;
                bptr->rd_prev = pptr;
                pptr->rd_next = bptr;

                /* Prepare to wait until item is processed */

                recvclr();
                resume(rdptr->rd_comproc);

                /* Block to wait for message */

                bptr = (struct rdbuff *)receive();
                break;

        /* Delete the remote disk (entirely remove it) */

        case RDS_CTL_DEL:

                /* Handcraft a message for the server that requests     */
                /*      deleting the disk with the specified ID         */

                msg.rd_type = htons(RD_MSG_DREQ);/* Request deletion    */
                msg.rd_status = htons(0);
                msg.rd_seq = 0; /* rdscomm will insert sequence # later */
                to = msg.rd_id;
                memset(to, NULLCH, RD_IDLEN);   /* initialize to zeroes */
```

```
                from = rdptr->rd_id;
                while ( (*to++ = *from++) != NULLCH ) { /* copy ID      */
                        ;
                }

                /* Send message and receive response */

                retval = rdscomm((struct rd_msg_hdr *)&msg,
                                        sizeof(struct rd_msg_dreq),
                         (struct rd_msg_hdr *)&resp,
                                        sizeof(struct rd_msg_dres),
                                        rdptr);

                /* Check response */

                if (retval == SYSERR) {
                        return SYSERR;
                } else if (retval == TIMEOUT) {
                        kprintf("Timeout during remote file delete\n\r");
                        return SYSERR;
                } else if (ntohs(resp.rd_status) != 0) {
                        return SYSERR;
                }

                /* Close local device */

                return rdsClose(devptr);

        default:
                kprintf("rfsControl: function %d not valid\n\r", func);
                return SYSERR;
        }

        return OK;
}
```

The code for each function should seem familiar. The code to delete an entire disk is similar to the code in *rdsOpen* — it creates a message for the server and uses *rdscomm* to send the message. The code to synchronize disk writes is similar to the code in *rdsRead* — it creates a request, enqueues the request, and calls *receive* to wait for a response. Once the response arrives, *rdsControl* invokes *rdsClose* to close the local device, and returns to its caller.

## 18.15 Allocating A Disk Buffer (rdsbufalloc)

As we have seen, driver functions call *rdsbufalloc* when they need to allocate a buffer. File *rdsbufalloc.c* contains the code:

```
/* rdsbufalloc.c  -  rdsbufalloc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsbufalloc - allocate a buffer from the free list or the cache
 *------------------------------------------------------------------------
 */
struct rdbuff *rdsbufalloc (
          struct rdscblk *rdptr         /* ptr to device control block  */
        )
{
        struct  rdbuff  *bptr;          /* ptr to a buffer              */
        struct  rdbuff  *pptr;          /* ptr to previous buffer       */
        struct  rdbuff  *nptr;          /* ptr to next buffer           */

        /* Wait for an available buffer */

        wait(rdptr->rd_availsem);

        /* If free list contains a buffer, extract it */

        bptr = rdptr->rd_free;

        if ( bptr != (struct rdbuff *)NULL ) {
                rdptr->rd_free = bptr->rd_next;
                return bptr;
        }

        /* Extract oldest item in cache that has ref count zero (at     */
        /*   least one such entry must exist because the semaphore      */
        /*   had a nonzero count)                                       */

        bptr = rdptr->rd_ctprev;
        while (bptr != (struct rdbuff *) &rdptr->rd_chnext) {
                if (bptr->rd_refcnt <= 0) {

                        /* Remove from cache and return to caller */
```

```
                        pptr = bptr->rd_prev;
                        nptr = bptr->rd_next;
                        pptr->rd_next = nptr;
                        nptr->rd_prev = pptr;
                        return bptr;
                }
                bptr = bptr->rd_prev;
        }
        panic("Remote disk cannot find an available buffer");
        return (struct rdbuff *)SYSERR;
}
```

Recall that a semaphore counts available buffers either on the free list or in the cache with a reference count of zero. After waiting on the semaphore, *rdsbufalloc* knows that a buffer will exist in one of the two places. It checks the free list first. If the free list is not empty, *rdsbufalloc* extracts the first buffer and returns it. If the free list is empty, *rdsbufalloc* searches the cache for an available buffer, extracts the buffer, and returns it to the caller. If the search completes without finding an available buffer, the count of the semaphore is incorrect, and *rdsbufalloc* calls *panic* to halt the system.

## 18.16 The Upper–Half Close Function (rdsClose)

A process invokes *close* to close the remote disk device and stop all communication. File *rdsClose.c* contains the code:

```
/* rdsClose.c  -  rdsClose */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsClose - Close a remote disk device
 *------------------------------------------------------------------------
 */
devcall rdsClose (
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  rdscblk *rdptr;         /* ptr to control block entry   */
        struct  rdbuff  *bptr;          /* ptr to buffer on a list      */
        struct  rdbuff  *nptr;          /* ptr to next buff on the list */
        int32   nmoved;                 /* number of buffers moved      */

        /* Device must be open */
```

```
        rdptr = &rdstab[devptr->dvminor];
        if (rdptr->rd_state != RD_OPEN) {
                return SYSERR;
        }

        /* Request queue must be empty */

        if (rdptr->rd_rhnext != (struct rdbuff *)&rdptr->rd_rtnext) {
                return SYSERR;
        }

        /* Move all buffers from the cache to the free list */

        bptr = rdptr->rd_chnext;
        nmoved = 0;
        while (bptr != (struct rdbuff *)&rdptr->rd_ctnext) {
                nmoved++;

                /* Unlink buffer from cache */

                nptr = bptr->rd_next;
                (bptr->rd_prev)->rd_next = nptr;
                nptr->rd_prev = bptr->rd_prev;

                /* Insert buffer into free list */

                bptr->rd_next = rdptr->rd_free;

                rdptr->rd_free = bptr;
                bptr->rd_status = RD_INVALID;

                /* Move to next buffer in the cache */

                bptr = nptr;
        }

        /* Set the state to indicate the device is closed */

        rdptr->rd_state = RD_FREE;
        return OK;
}
```

To close a remote disk device, all buffers must be moved back to the free list (re-creating the conditions immediately following initialization) and the state field in the control block must be assigned *RD_FREE*. Our implementation removes buffers from the cache, but does not handle the request list. Instead, we require a user to wait until all requests have been satisfied and the request list is empty before calling *rdsClose*. The synchronization function, *RDS_CTL_SYNC*,† provides a way to wait for the request queue to drain.

## 18.17 The Lower–Half Communication Process (rdsprocess)

In the example implementation, each remote disk device has its own control block, its own set of disk buffers, and its own remote communication process. Thus, a given remote disk process only needs to handle requests from a single queue. Although the code may seem long and filled with details, the general algorithm is straightforward: repeatedly wait on the request semaphore, examine the type of the request at the head of the queue, and either perform a *read*, a *write*, or a *synchronization* operation. File *rdsprocess.c* contains the code:

```
/* rdsprocess.c  -  rdsprocess */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsprocess - high-priority background process that repeatedly extracts
 *              an item from the request queue and sends the request to
 *              the remote disk server
 *------------------------------------------------------------------------
 */
void    rdsprocess (
          struct rdscblk    *rdptr      /* ptr to device control block  */
        )
{
        struct  rd_msg_wreq msg;        /* message to be sent           */
                                        /*  (includes data area)        */
        struct  rd_msg_rres resp;       /* buffer to hold response      */
                                        /*  (includes data area)        */
        int32   retval;                 /* return value from rdscomm    */
        char    *idto;                  /* ptr to ID string copy        */
        char    *idfrom;                /* ptr into ID string           */
        struct  rdbuff  *bptr;          /* ptr to buffer at the head of */
                                        /*   the request queue          */
        struct  rdbuff  *nptr;          /* ptr to next buffer on the    */
                                        /*   request queue              */
```

†The synchronization code is found in file *rdsControl.c* on page 396.

```
struct  rdbuff  *pptr;          /* ptr to previous buffer       */
struct  rdbuff  *qptr;          /* ptr that runs along the      */
                                /*   request queue              */
int32   i;                      /* loop index                   */

while (TRUE) {                  /* do forever */

    /* Wait until the request queue contains a node */
    wait(rdptr->rd_reqsem);
    bptr = rdptr->rd_rhnext;

    /* Use operation in request to determine action */

   switch (bptr->rd_op) {

   case RD_OP_READ:

        /* Build a read request message for the server */

        msg.rd_type = htons(RD_MSG_RREQ);       /* read request */
        msg.rd_status = htons(0);
        msg.rd_seq = 0;         /* rdscomm fills in an entry    */
        idto = msg.rd_id;
        memset(idto, NULLCH, RD_IDLEN);/* initialize ID to zero */
        idfrom = rdptr->rd_id;
        while ( (*idto++ = *idfrom++) != NULLCH ) { /* copy ID  */
                ;
        }

        /* Send the message and receive a response */

        retval = rdscomm((struct rd_msg_hdr *)&msg,
                                sizeof(struct rd_msg_rreq),
                        (struct rd_msg_hdr *)&resp,
                                sizeof(struct rd_msg_rres),
                         rdptr );

        /* Check response */

        if ( (retval == SYSERR) || (retval == TIMEOUT) ||
                        (ntohs(resp.rd_status) != 0) ) {
                panic("Failed to contact remote disk server");
        }

        /* Copy data from the reply into the buffer */
```

```
for (i=0; i<RD_BLKSIZ; i++) {
        bptr->rd_block[i] = resp.rd_data[i];
}

/* Unlink buffer from the request queue */

nptr = bptr->rd_next;
pptr = bptr->rd_prev;
nptr->rd_prev = bptr->rd_prev;
pptr->rd_next = bptr->rd_next;

/* Insert buffer in the cache */

pptr = (struct rdbuff *) &rdptr->rd_chnext;
nptr = pptr->rd_next;
bptr->rd_next = nptr;
bptr->rd_prev = pptr;
pptr->rd_next = bptr;
nptr->rd_prev = bptr;

/* Initialize reference count */

bptr->rd_refcnt = 1;

/* Signal the available semaphore */

signal(rdptr->rd_availsem);

/* Send a message to waiting process */

send(bptr->rd_pid, (uint32)bptr);

/* If other processes are waiting to read the  */
/*   block, notify them and remove the request */

qptr = rdptr->rd_rhnext;
while (qptr != (struct rdbuff *)&rdptr->rd_rtnext) {
        if (qptr->rd_blknum == bptr->rd_blknum) {
                bptr->rd_refcnt++;
                send(qptr->rd_pid,(uint32)bptr);

                /* Unlink request from queue    */

                pptr = qptr->rd_prev;
                nptr = qptr->rd_next;
                pptr->rd_next = bptr->rd_next;
```

```
                        nptr->rd_prev = bptr->rd_prev;

                        /* Move buffer to the free list */

                        qptr->rd_next = rdptr->rd_free;
                        rdptr->rd_free = qptr;
                        signal(rdptr->rd_availsem);
                        break;
                }
                qptr = qptr->rd_next;
        }
        break;

    case RD_OP_WRITE:

        /* Build a write request message for the server */

        msg.rd_type = htons(RD_MSG_WREQ);       /* write request*/
        msg.rd_blk = bptr->rd_blknum;
        msg.rd_status = htons(0);
        msg.rd_seq = 0;         /* rdscomm fills in an entry    */
        idto = msg.rd_id;
        memset(idto, NULLCH, RD_IDLEN);/* initialize ID to zero */
        idfrom = rdptr->rd_id;
        while ( (*idto++ = *idfrom++) != NULLCH ) { /* copy ID  */
                ;
        }
        for (i=0; i<RD_BLKSIZ; i++) {
                msg.rd_data[i] = bptr->rd_block[i];
        }

        /* Unlink buffer from request queue */

        nptr = bptr->rd_next;
        pptr = bptr->rd_prev;
        pptr->rd_next = nptr;
        nptr->rd_prev = pptr;

        /* Insert buffer in the cache */

        pptr = (struct rdbuff *) &rdptr->rd_chnext;
        nptr = pptr->rd_next;
        bptr->rd_next = nptr;
        bptr->rd_prev = pptr;
        pptr->rd_next = bptr;
        nptr->rd_prev = bptr;
```

```
                /* Declare that buffer is eligible for reuse */

                bptr->rd_refcnt = 0;
                signal(rdptr->rd_availsem);

                /* Send the message and receive a response */

                retval = rdscomm((struct rd_msg_hdr *)&msg,
                                        sizeof(struct rd_msg_wreq),
                                (struct rd_msg_hdr *)&resp,
                                        sizeof(struct rd_msg_wres),
                                 rdptr );

                /* Check response */

                if ( (retval == SYSERR) || (retval == TIMEOUT) ||
                                (ntohs(resp.rd_status) != 0) ) {
                        panic("failed to contact remote disk server");
                }
                break;

            case RD_OP_SYNC:

                /* Send a message to the waiting process */

                send(bptr->rd_pid, OK);

                /* Unlink buffer from the request queue */

                nptr = bptr->rd_next;
                pptr = bptr->rd_prev;
                nptr->rd_prev = bptr->rd_prev;
                pptr->rd_next = bptr->rd_next;

                /* Insert buffer into the free list */

                bptr->rd_next = rdptr->rd_free;
                rdptr->rd_free = bptr;
                signal(rdptr->rd_availsem);
                break;
            }
        }
}
```

When examining the code, remember that the remote disk process has higher priority than any application process. Thus, the code does not need to disable interrupts or use a mutual exclusion semaphore when accessing the request queue, cache, or free list. However, *rdsprocess* must leave all data structures in a valid state before using *rdscomm* to exchange messages with the server because message reception blocks the calling process (which means other processes can run). In the case of a *read* operation, *rdsprocess* leaves the buffer on the request queue until the request can be satisfied. In the case of a *write* operation, *rdsprocess* extracts a copy of the data and moves the buffer to the cache before calling *rdscomm*.

## 18.18 Perspective

Conceptually, a remote disk system only needs to provide two basic operations: *read* a block and *write* a block. In practice, however, the issues of synchrony, caching, and sharing dominate the design. Our example simplifies most design decisions because we assume a single Xinu system acts as a client. Thus, the client code manages its local cache, and does not need to coordinate with other Xinu systems. Similarly, the lack of sharing simplifies the question of synchrony: the client only needs local information to enforce *last-write semantics*.

If the system is extended to permit multiple Xinu systems to share a disk, the entire design must change. A given client cannot cache blocks unless the client coordinates with the server. Furthermore, *last-write semantics* must be enforced across all systems, which means *read* operations need a centralized mechanism to insure that they occur in order. The point is:

> *Extending the remote disk system to include sharing across multiple Xinu systems will result in significant changes to the structure of the system.*

## 18.19 Summary

We considered the design of a remote disk system in which an application can *read* and *write* disk blocks, and the driver uses a network to communicate with a remote server that performs the operation. The driver views a disk as an array of randomly accessible data blocks, and does not provide files, directories, or any index techniques to speed searching. Reading consists of copying data from a specified block on disk into memory; writing consists of copying data from memory onto a specified disk block.

Driver code is divided into upper-half functions that are called by application processes and a lower half that executes as a separate process. Input is synchronous; a process blocks until a request can be satisfied. Output is asynchronous; the driver accepts an outgoing data block, enqueues the request, and returns to the caller immediate-

ly without blocking. A process can use the *control* function to flush previous writes to disk.

The driver uses three main data structures: a queue of requests, a cache of recently used blocks, and a free list. Although it relies on caching, the driver guarantees last-write semantics.

## EXERCISES

**18.1** Redesign the implementation to keep the buffers separate from the nodes used on the request list and the cache (i.e., define nodes for each list and arrange for each node to have a pointer to a buffer). What are the advantages and disadvantages of each approach?

**18.2** Redesign the remote disk system to use a "buffer exchange" paradigm in which applications and driver functions share a single buffer pool. To write a disk block, arrange for an application to allocate a buffer, fill the buffer, and pass the buffer when calling *write*. Have *read* return a buffer pointer which the application must free once the data has been extracted.

**18.3** It is possible to configure a system with multiple remote disk devices. Modify the code in *rdsOpen* to check each open remote disk device to insure that a disk ID is unique.

**18.4** Build a version of a remote disk system that does not use a cache, and measure the difference in performance of the two versions.

**18.5** Should requests from high-priority processes take precedence over requests from low-priority processes? Explain why or why not.

**18.6** Investigate other algorithms like the "elevator" algorithm that can be used to order disk requests for an electro-mechanical disk.

**18.7** Verify that a request to "synchronize" will not return until all pending requests have been satisfied. Is there a bound on the time it can be delayed?

# NOTES

# Chapter Contents

# 19

# File Systems

*Filing is concerned with the past; anything you actually need to see again has to do with the future.*

— Katharine Whitehorn

The previous chapter discusses disk devices, and describes a hardware interface that allows the system to read and write individual blocks. Although disks have the advantage of providing long-term, non-volatile storage, the block-oriented interface is cumbersome.

This chapter introduces the file system abstraction. It shows how an operating system manages a set of dynamically changing file objects, and how the system maps files onto the underlying disk hardware.

## 19.1 What Is A File System?

A *file system* consists of software that manages permanent data objects whose values persist longer than the processes that create and use them. Permanent data is kept in *files*, which are stored on secondary storage devices, primarily disks. Files are organized into *directories* (also called *folders*). Conceptually, each file consists of a sequence of data objects (e.g., a sequence of integers). The file system provides operations that *create* or *delete* a file, *open* a file given its name, *read* the next object from an open file, *write* an object onto an open file, or *close* a file. If a file system allows random access, the file interface also provides a way a process can *seek* to a specified location in a file.

Many file systems do more than manipulate individual files on secondary storage — they provide an abstract namespace and high-level operations to manipulate objects in that space. The file namespace consists of the set of valid file names. A namespace can be as simple as "the set of strings formed from at least one but fewer than nine alphabetic characters," or as complex as "the set of strings that form a valid encoding of the network, machine, user, subdirectory, and file identifiers in a specified syntax." In some systems, the syntax of names in the abstract space conveys information about their type (e.g., text files end in ".txt"). In others, names give information about the organization of the file system (e.g., a file name that begins with the string "M1_d0:" might reside on disk 0 of machine 1). We will defer a discussion of file naming to Chapter 21, and concentrate on file access.

## 19.2 An Example Set Of File Operations

Our example system uses a straightforward approach motivated by a desire to unify the interface between devices and files and to keep the file system software small. File semantics are taken from Unix according to the following principle:

> *The file system considers each file to be a sequence of zero or more bytes; any further structure must be enforced by application programs that use the file.*

Treating a file as a stream of bytes has several advantages. First, the file system does not impose a type on the file and does not need to distinguish among file types. Second, the code is small because a single set of file system functions suffices for all files. Third, the file semantics can be applied to devices and services as well as to conventional files. Fourth, application programs can choose an arbitrary structure for data without changing the underlying system. Finally, file contents are independent of the processor or memory (e.g., an application may need to distinguish among a 32-bit and 64-bit integer stored in a file, but the file system does not).

Our system will use exactly the same high-level operations for files that are used for devices. Thus, the file system will support *open*, *close*, *read*, *write*, *putc*, *getc*, *seek*, *init*, and *control* functions. When applied to conventional files, the operations produce the following results. *Init* initializes data structures at startup. *Open*ing a named file connects an executing process with the data on disk, and establishes a pointer to the first byte. Operations *getc* and *read* retrieve data from the file and advance the pointer; *getc* retrieves one byte, and *read* can retrieve multiple bytes. Operations *putc* and *write* change bytes in the file and move the pointer along, extending the file length if new data is written beyond the end; *putc* changes one byte, and *write* can change multiple bytes. The *seek* operation moves the pointer to a specified byte position in the file; the first byte is at position zero. Finally, *close* detaches the running process from the file, leaving the data in the file on permanent storage.

## 19.3 Design Of A Local File System

A file is said to be *local* to a given computer if the file resides on a disk that is connected to the computer. The design of software that manages such files is nontrivial; it has been the subject of much research. Although the file operations may seem straightforward, complexity arises because files are *dynamic*. That is, a single disk can hold multiple files, and a given file can grow arbitrarily large (until disk space is exhausted). To permit dynamic file growth, a system cannot pre-allocate disk blocks for a file. Thus, dynamic data structures are needed.

A second form of complexity arises from concurrency. To what extent should the system support concurrent file operations? Large systems usually allow arbitrary numbers of processes to read and write arbitrary numbers of files concurrently. The chief difficulty with multiple access lies in specifying exactly what it means to have multiple processes writing and reading a file at the same time. When will data become available for reading? If two processes attempt to write to the same data byte in the file, which will be accepted? Can a process lock pieces of a file to avoid interference?

The generality of allowing multiple processes to read and write a file is usually not necessary on small embedded systems. Thus, to limit the software complexity and make better use of disk space, small systems can constrain the ways in which files can be accessed. They may limit the number of files that a given process can access simultaneously, or limit the number of processes that can access a given file simultaneously.

Our goal is to design efficient, compact file system software that allows processes to create and extend files dynamically without incurring unnecessary overhead. As a compromise between generality and efficiency, we will allow a process to open an arbitrary number of files until resources are exhausted. However, the system limits a file to one active open. That is, if a file is open, successive requests to open it will fail until the file has been closed. Each file has a mutual exclusion semaphore to guarantee that only one process at a time can attempt to write a byte to the file, read a byte from the file, or change the current file position. Furthermore, the directory has a mutual exclusion semaphore to guarantee that only one process at a time can attempt to create a file or otherwise change a directory entry. Although concurrency requires attention to detail, the most significant consequence of our design arises from its support for dynamic file growth: data structures will be needed to allocate space on a disk dynamically. The next section explains the data structures used.

## 19.4 Data Structures For The Xinu File System

To support dynamic growth and random access, the Xinu file system allocates disk blocks dynamically and uses an *index mechanism* to locate the data in a given file quickly. The design partitions a disk into three separate areas as Figure 19.1 illustrates: a *directory*, an *index area*, and a *data area*.


**Figure 19.1** Illustration of a disk partitioned into three areas for the Xinu file system.

The first sector of the disk holds a directory that contains a list of file names along with a pointer to the list of index blocks for the file. The directory also contains two other pointers: one to a list of free (unused) index blocks and another to a list of free data blocks. The directory entry for a file also contains an integer that gives the current size of the file measured in bytes.

Following the directory, the disk contains an index area that holds a set of *index blocks*, abbreviated *i-blocks*. Each file has its own index, which consists of a singly-linked list of index blocks. Initially, all index blocks are linked onto a free list from which the system allocates one as needed; index blocks are only returned to the free list if a file is truncated or deleted.

Following the index area, remaining blocks of the disk comprise a data area. Each block in the data area is referred to as a *data block*, abbreviated *d-block*, because a block contains data that has been stored in a file. Once a data block has been allocated to a file, the block only can contain data. A data block does not contain pointers to other data blocks, nor does it contain information that relates the block to the file of which it is a part; all such information resides in the file's index.

Similar to index blocks, when a disk is initialized, the data blocks are linked onto a free list. The file system allocates data blocks from the free list as needed, and returns data blocks to the free list when a file is truncated or deleted.

Figure 19.2 illustrates the conceptual data structure used for a Xinu file system. The figure is not drawn to scale: in practice a data block is much larger than an index block and occupies one physical disk block.

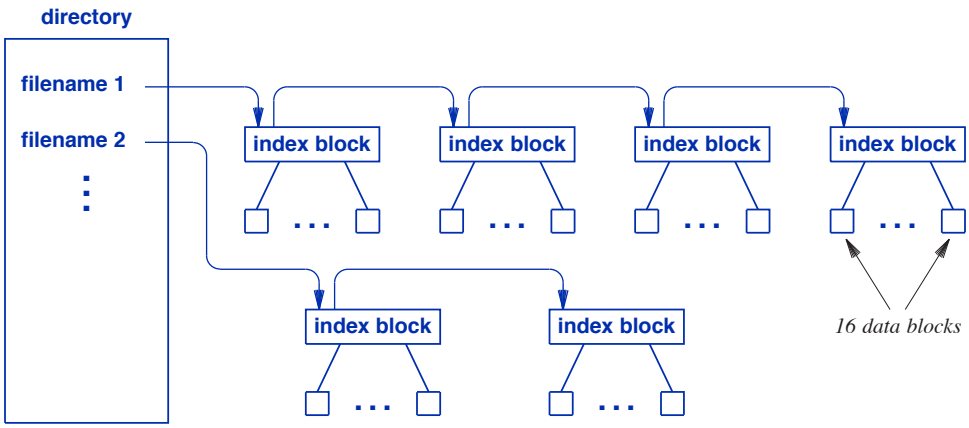



**Figure 19.2** Illustration of the Xinu file system, where each file consists of a linked list of index blocks that each contain pointers to data blocks.

The important idea is that the data structure illustrated in the figure resides on disk. We will see that at any given time, only a few pieces of the structure are present in memory — the file system must create and maintain an index without reading the structure into memory.

## 19.5 Implementation Of The Index Manager

Conceptually index blocks form a randomly-accessible array that is mapped onto a contiguous area of the disk. Thus, index blocks are numbered from zero through *K*, and the software uses the number to refer to a given index block. Because an index block is smaller than a physical disk block, the system stores seven index blocks into each physical block, and the software handles the details of reading and writing an individual index block.

Because the underlying hardware can only transfer an entire disk block at a time, the file system cannot transfer an individual index block without transferring others that reside in the same physical disk block. Therefore, to write an index block, the software must read the entire physical disk block in which it resides, copy the new index block into the correct position, and write the resulting physical block back to disk. Similarly, to read an index block, the software must read the physical disk block in which it resides, and then extract the index block.

Before we examine the code to handle index blocks, we need to understand basic definitions. File *lfilesys.h* defines constants and data structures used throughout the local file system, including struct *lfiblk* which defines the contents of an index block.

```
/* lfilesys.h - ib2sect, ib2disp */

/************************************************************************/
/*                                                                      */
/*              Local File System Data Structures                       */
/*                                                                      */
/*   A local file system uses a random-access disk composed of 512-byte */
/* sectors numbered 0 through N-1.  We assume disk hardware can read or */
/* write any sector at random, but must transfer an entire sector.      */
/* Thus, to write a few bytes, the file system must read the sector,    */
/* replace the bytes, and then write the sector back to disk.  Xinu's   */
/* local file system divides the disk as follows: sector 0 is a         */
/* directory, the next K sectors constitute an index area, and the      */
/* remaining sectors comprise a data area. The data area is easiest to  */
/* understand: each sector holds one data block (d-block) that stores   */
/* contents from one of the files (or is on a free list of unused data  */
/* blocks).  We think of the index area as holding an array of index    */
/* blocks (i-blocks) numbered 0 through I-1.  A given sector in the     */
/* index area holds 7 of the index blocks, which are each 72 bytes      */
/* long.  Given an i-block number, the file system must calculate the   */
/* disk sector in which the i-block is located and the byte offset      */
/* within the sector at which the i-block resides.  Internally, a file  */
/* is known by the i-block index of the first i-block for the file.     */
/* The directory contains a list of file names and the  i-block number  */
/* of the first i-block for the file.  The directory also holds the     */
/* i-block number for a list of free i-blocks and a data block number   */
/* of the first data block on a list of free data blocks.               */
/*                                                                      */
/************************************************************************/

#ifndef Nlfl
#define Nlfl    1
#endif

/* Use the remote disk device if no disk is defined (file system  */
/*  *assumes* the underlying disk has a block size of 512 bytes)  */

#ifndef LF_DISK_DEV
#define LF_DISK_DEV     SYSERR
#endif

#define LF_MODE_R       F_MODE_R        /* mode bit for "read"          */
#define LF_MODE_W       F_MODE_W        /* mode bit for "write"         */
#define LF_MODE_RW      F_MODE_RW       /* mode bits for "read or write"*/
#define LF_MODE_O       F_MODE_O        /* mode bit for "old"           */
```

```
#define LF_MODE_N       F_MODE_N        /* mode bit for "new"           */

#define LF_BLKSIZ       512             /* assumes 512-byte disk blocks */
#define LF_NAME_LEN     16              /* length of name plus null     */
#define LF_NUM_DIR_ENT  20              /* num. of files in a directory */

#define LF_FREE         0               /* slave device is available    */
#define LF_USED         1               /* slave device is in use       */

#define LF_INULL        (ibid32) -1     /* index block null pointer     */
#define LF_DNULL        (dbid32) -1     /* data block null pointer      */
#define LF_IBLEN        16              /* data block ptrs per i-block  */
#define LF_IDATA        8192            /* bytes of data indexed by a   */
                                        /*  single index block          */
#define LF_IMASK        0x00001fff      /* mask for the data indexed by */
                                        /*  a single index block (i.e., */
                                        /*  bytes 0 through 8191).      */
#define LF_DMASK        0x000001ff      /* mask for the data in a data  */
                                        /*  block  (0 through 511)      */

#define LF_AREA_IB      1               /* first sector of i-blocks     */
#define LF_AREA_DIR     0               /* first sector of directory    */

/* Structure of an index block on disk */

struct  lfiblk          {               /* format of index block        */
        ibid32          ib_next;        /* address of next index block  */
        uint32          ib_offset;      /* first data byte of the file  */
                                        /*  indexed by this i-block     */
        dbid32          ib_dba[LF_IBLEN];/* ptrs to data blocks indexed */
};

/* Conversion functions below assume 7 index blocks per disk block */

/* Conversion between index block number and disk sector number */

#define ib2sect(ib)     (((ib)/7)+LF_AREA_IB)

/* Conversion between index block number and the relative offset within */
/*      a disk sector                                                   */

#define ib2disp(ib)     (((ib)%7)*sizeof(struct lfiblk))


/* Structure used in each directory entry for the local file system */
```

```
struct  ldentry {                       /* description of entry for one */
                                        /*  file in the directory       */
        uint32  ld_size;                /* curr. size of file in bytes  */
        ibid32  ld_ilist;               /* ID of first i-block for file */
                                        /*   or IB_NULL for empty file  */
        char    ld_name[LF_NAME_LEN];   /* null-terminated file name    */
};

/* Structure of a data block when on the free list on disk */

struct  lfdbfree {
        dbid32  lf_nextdb;              /* next data block on the list  */
        char    lf_unused[LF_BLKSIZ - sizeof(dbid32)];
};

/* Format of the file system directory, either on disk or in memory */

#pragma pack(2)
struct  lfdir   {                       /* entire directory on disk     */
        dbid32  lfd_dfree;              /* list of free d-blocks on disk*/
        ibid32  lfd_ifree;              /* list of free i-blocks on disk*/
        int32   lfd_nfiles;             /* current number of files      */
        struct  ldentry lfd_files[LF_NUM_DIR_ENT]; /* set of files      */
        char    padding[20];            /* unused chars in directory blk*/
};
#pragma pack()

/* Global data used by local file system */

struct  lfdata  {                       /* local file system data       */
        did32   lf_dskdev;              /* device ID of disk to use     */
        sid32   lf_mutex;               /* mutex for the directory and  */
                                        /*  index/data free lists       */
        struct  lfdir   lf_dir;         /* In-memory copy of directory  */
        bool8   lf_dirpresent;          /* True when directory is in    */
                                        /*  memory (first file is open) */
        bool8   lf_dirdirty;            /* Has the directory changed?   */
};

/* Control block for local file pseudo-device */

struct  lflcblk {                       /* Local file control block     */
                                        /*  (one for each open file)    */
        byte    lfstate;                /* Is entry free or used        */
```

```
        did32   lfdev;                  /* device ID of this device     */
        sid32   lfmutex;                /* Mutex for this file          */
        struct  ldentry *lfdirptr;      /* Ptr to file's entry in the   */
                                        /*  in-memory directory         */
        int32   lfmode;                 /* mode (read/write/both)       */
        uint32  lfpos;                  /* Byte position of next byte   */
                                        /*   to read or write           */
        char    lfname[LF_NAME_LEN];    /* Name of the file             */
        ibid32  lfinum;                 /* ID of current index block in */
                                        /*   lfiblock or LF_INULL       */
        struct  lfiblk  lfiblock;       /* In-mem copy of current index */
                                        /*   block                      */
        dbid32  lfdnum;                 /* Number of current data block */
                                        /*   in lfdblock or LF_DNULL    */
        char    lfdblock[LF_BLKSIZ];    /* in-mem copy of current data  */
                                        /*   block                      */
        char    *lfbyte;                /* Ptr to byte in lfdblock or   */
                                        /*  address one beyond lfdblock */
                                        /*   if current file pos lies   */
                                        /*   outside lfdblock           */
        bool8   lfibdirty;              /* Has lfiblock changed?        */
        bool8   lfdbdirty;              /* Has lfdblock changed?        */
};

extern  struct  lfdata  Lf_data;
extern  struct  lflcblk lfltab[];

/* Control functions */

#define LF_CTL_DEL      F_CTL_DEL       /* Delete a file                */
#define LF_CTL_TRUNC    F_CTL_TRUNC     /* Truncate a file              */
#define LF_CTL_SIZE     F_CTL_SIZE      /* Obtain the size of a file    */
```

As the file shows, each index block contains a pointer to the next index block, an *offset* that specifies the lowest position in the file indexed by the block, and an array of sixteen pointers to data blocks. That is, each entry in the array gives the physical disk sector number of a data block. Because a sector is 512 bytes long, a single index block indexes sixteen 512-byte blocks or 8,192 bytes of data.

How does the software know where to find an index block given its address? Index blocks are contiguous, and occupy contiguous disk sectors starting at sector *LF_AREA_IB*. In our design, the directory occupies disk block zero, which means that the index area starts at sector one. Thus, index blocks zero through seven lie in sector one, eight through fifteen lie in sector two, and so on. Inline function *ib2sect* converts an index block number into the correct sector number, and inline function *ib2disp* con-

verts an index block number to a byte displacement within a physical disk block. Both functions can be found in file *lfilesys.h* above.

## 19.6 Clearing An Index Block (lfibclear)

Whenever it allocates an index block from the free list, the file system must read the index block into memory and clear the block to remove old information. In particular, all data block pointers must be set to a null value, so they will not be confused with valid pointers. Furthermore, the offset in the index block must be assigned the appropriate offset in the file. Function *lfibclear* clears an index block; file *lfibclear.c* contains the code.

```
/* lfibclear.c - lfibclear */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfibclear  --  clear an in-core copy of an index block
 *------------------------------------------------------------------------
 */
void    lfibclear(
          struct lfiblk *ibptr,         /* address of i-block in memory */
          int32         offset          /* file offset for this i-block */
        )
{
        int32   i;                      /* indexes through array        */

        ibptr->ib_offset = offset;      /* assign specified file offset */
        for (i=0 ; i<LF_IBLEN ; i++) {  /* clear each data block pointer*/
                ibptr->ib_dba[i] = LF_DNULL;
        }
        ibptr->ib_next = LF_INULL;      /* set next ptr to null         */
}
```

## 19.7 Retrieving An Index Block (lfibget)

To read an index block into memory, the system must map the index block number to a physical disk block address, read the physical disk block, and copy the appropriate area from the physical block into the specified memory location. File *lfibget.c* contains the code, which uses inline function *ib2sect* to convert the index block number to a disk sector, and function *ib2disp* to compute the location of the index block within the disk sector.

```
/* lfibget.c - lfibget */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfibget  --  get an index block from disk given its number (assumes
 *                      mutex is held)
 *------------------------------------------------------------------------
 */
void    lfibget(
          did32         diskdev,        /* device ID of disk to use     */
          ibid32        inum,           /* ID of index block to fetch   */
          struct lfiblk *ibuff          /* buffer to hold index block   */
        )
{
        char    *from, *to;             /* pointers used in copying     */
        int32   i;                      /* loop index used during copy  */
        char    dbuff[LF_BLKSIZ];       /* ibuff to hold disk block     */

        /* Read disk block that contains the specified index block */

        read(diskdev, dbuff, ib2sect(inum));

        /* Copy specified index block to caller's ibuff */

        from = dbuff + ib2disp(inum);
        to = (char *)ibuff;
        for (i=0 ; i<sizeof(struct lfiblk) ; i++)
                *to++ = *from++;
        return;
}
```

## 19.8 Storing An Index Block (lfibput)

Storing an index block is more complicated than retrieving one because the code must first read the appropriate disk sector, copy the index block into the appropriate area, and then write the sector back to disk. File *lfibput.c* contains the code, which uses the same inline functions as *lfibget*:

```
/* lfibput.c - lfibput */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfibput  --  write an index block to disk given its ID (assumes
 *                      mutex is held)
 *------------------------------------------------------------------------
 */
status  lfibput(
          did32         diskdev,        /* ID of disk device            */
          ibid32        inum,           /* ID of index block to write   */
          struct lfiblk *ibuff          /* buffer holding the index blk */
        )
{
        dbid32  diskblock;              /* ID of disk sector (block)    */
        char    *from, *to;             /* pointers used in copying     */
        int32   i;                      /* loop index used during copy  */
        char    dbuff[LF_BLKSIZ];       /* temp. buffer to hold d-block */

        /* Compute disk block number and offset of index block */

        diskblock = ib2sect(inum);
        to = dbuff + ib2disp(inum);
        from = (char *)ibuff;

        /* Read disk block */

        if (read(diskdev, dbuff, diskblock) == SYSERR) {
                return SYSERR;
        }

        /* Copy index block into place */

        for (i=0 ; i<sizeof(struct lfiblk) ; i++) {
                *to++ = *from++;
        }

        /* Write the block back to disk */

        write(diskdev, dbuff, diskblock);
        return OK;
}
```

## 19.9 Allocating An Index Block From The Free List (lfiballoc)

The file system allocates an index block from the free list whenever it needs to extend the index for a file. Function *lfiballoc* obtains the next free index block and returns its identifier. The code, which is found in file *lfiballoc.c*, assumes that a copy of the directory for the file system has been read into memory and placed in global variable *Lf_data.lf_dir*.

```
/* lfiballoc.c  -  lfiballoc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfiballoc  -  allocate a new index block from free list on disk
 *                      (assumes directory mutex held)
 *------------------------------------------------------------------------
 */
ibid32  lfiballoc (void)
{
        ibid32  ibnum;          /* ID of next block on the free list    */
        struct  lfiblk  iblock; /* buffer to hold index block           */

        /* Get ID of first index block on free list */

        ibnum = Lf_data.lf_dir.lfd_ifree;
        if (ibnum == LF_INULL) {        /* ran out of free index blocks */
                panic("out of index blocks");
        }
        lfibget(Lf_data.lf_dskdev, ibnum, &iblock);

        /* Unlink index block from the directory free list */

        Lf_data.lf_dir.lfd_ifree = iblock.ib_next;

        /* Write a copy of the directory to disk after the change */

        write(Lf_data.lf_dskdev, (char *) &Lf_data.lf_dir, LF_AREA_DIR);
        Lf_data.lf_dirdirty = FALSE;

        return ibnum;
}
```

## 19.10 Allocating A Data Block From The Free List (lfdballoc)

Because an index block contains a "next" pointer field, linking them into a free list is straightforward. For data blocks, however, the free list is less obvious because a data block does not usually contain a pointer field. The Xinu design uses a singly-linked free list, which means that only one pointer is needed. When a data block is on the free list, the system uses the first four bytes of the data block to store a pointer to the next block on the list. Structure *lfdbfree*, found in file *lfilesys.h* above, defines the format of a block on the free list. Whenever it extracts a data block from the free list, the file system uses the structure definition. Of course, once a block has been removed from the free list and allocated to a file, the block is treated as an array of bytes.

Function *lfdballoc*, which allocates a data block from the free list and returns the block number, illustrates how the system uses struct *lfdbfree*. The code can be found in file *lfdballoc.c*.

```
/* lfdballoc.c  -  lfdballoc */

#include <xinu.h>

#define  DFILL  '+'                     /* char. to fill a disk block   */

/*------------------------------------------------------------------------
 * lfdballoc  -  allocate a new data block from free list on disk
 *                      (assumes directory mutex held)
 *------------------------------------------------------------------------
 */
dbid32  lfdballoc (
          struct lfdbfree *dbuff /* addr. of buffer to hold data block  */
        )
{
        dbid32  dnum;           /* ID of next d-block on the free list  */
        int32   retval;         /* return value                         */

        /* Get the ID of first data block on the free list */

        dnum = Lf_data.lf_dir.lfd_dfree;
        if (dnum == LF_DNULL) { /* ran out of free data blocks */
                panic("out of data blocks");
        }
        retval = read(Lf_data.lf_dskdev, (char *)dbuff, dnum);
        if (retval == SYSERR) {
                panic("lfdballoc cannot read disk block\n\r");
        }
```

```
        /* Unlink d-block from in-memory directory */

        Lf_data.lf_dir.lfd_dfree = dbuff->lf_nextdb;
        write(Lf_data.lf_dskdev, (char *)&Lf_data.lf_dir, LF_AREA_DIR);
        Lf_data.lf_dirdirty = FALSE;

        /* Fill data block to erase old data */

        memset((char *)dbuff, DFILL, LF_BLKSIZ);
        return dnum;
}
```

A corresponding function, *lfdbfree*, returns a block to the free list. The code can be found in file *lfdbfree.c*.

```
/* lfdbfree.c - lfdbfree */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfdbfree  --  free a data block given its block number (assumes
 *                      directory mutex is held)
 *------------------------------------------------------------------------
 */
status  lfdbfree(
          did32         diskdev,        /* ID of disk device to use     */
          dbid32        dnum            /* ID of data block to free     */
        )
{
        struct  lfdir   *dirptr;        /* pointer to directory         */
        struct  lfdbfree buf;           /* buffer to hold data block    */

        dirptr = &Lf_data.lf_dir;
        buf.lf_nextdb = dirptr->lfd_dfree;
        dirptr->lfd_dfree = dnum;
        write(diskdev, (char *)&buf,   dnum);
        write(diskdev, (char *)dirptr, LF_AREA_DIR);

        return OK;
}
```

To place a block on the free list, *lfdbfree* first makes the block point to the current free list, and then makes the current free list point to the block. Because a pointer has

been inserted, the block must be written to disk; because the free list in the directory has been changed, a copy of the directory must be written to disk.

## 19.11 Using The Device-Independent I/O Functions For Files

The file system software must establish connections between running processes and disk files to allow operations like *read* and *write* to be mapped onto the correct file. Exactly how the system performs this mapping depends on both the size and generality needed. To keep our system small, we will avoid introducing new functions by using the device switch mechanisms already in place.

Imagine that a set of file *pseudo-devices* has been added to the device switch table such that each pseudo-device can be used to control an open file. As with conventional devices, a pseudo-device has a set of driver functions that perform *read*, *write*, *getc*, *putc*, *seek*, and *close* operations. When a process opens a disk file, the file system searches for a currently unused pseudo-device, sets up the control block for the pseudo-device, and returns the ID of the pseudo-device to the caller. After the file has been opened, the process uses the device ID with operations *getc*, *read*, *putc*, *write*, and *seek*. The device switch maps each high-level operation to the appropriate driver function for file pseudo-devices exactly as it maps high-level operations onto device drivers for physical devices. Finally, when it finishes using a file, a process calls *close* to break the connection and make the pseudo-device available for use with another file. The details will become clear as we review the code.

Designing a pseudo-device driver is not unlike designing a device driver for a conventional hardware device. Just like other drivers, the pseudo-device driver creates a control block for each pseudo-device. The control block for a file pseudo-device uses struct *lflcblk*, which is defined in file *lfilesys.h*. Conceptually, the control block contains two types of items: fields that hold information about the pseudo-device and fields that hold information from the disk. Fields *lfstate* and *lfmode* are the former type: the state field specifies whether the device is currently in use, and the mode field specifies whether the file has been opened for reading, writing, or both. Fields *lfiblock* and *lfdblock* are of the latter type: when a file is being read or written they contain a copy of the index block and the data block for the current position in the file measured in bytes (which is given by field *lfpos*).

When a file is opened, the position (field *lfpos* in the control block) is assigned zero. As processes *read* or *write* data, the position increases. A process can call *seek* to move to an arbitrary position in the file, and *lfpos* is updated.

## 19.12 File System Device Configuration And Function Names

What interface should be used to open a file and allocate a pseudo-device for reading and writing? Because Xinu tends to map all functions into the device space, the local file system uses the approach of defining a master local file device, *LFILESYS*. Cal-

ling open on the *LFILESYS* device causes the system to allocate a pseudo-device and return the device ID of the pseudo-device. File pseudo-devices are named *LFILE0*, *LFILE1*, ..., but the names are used only in the configuration file. Figure 19.3 shows how the master and file pseudo-devices are configured.

```
/* Local File System master device type */

lfs: on disk
                -i lfsInit      -o lfsOpen      -c ioerr
                -r ioerr        -g ioerr        -p ioerr
                -w ioerr        -s ioerr        -n rfsControl
                -intr NULL

/* Local file pseudo-device type */

lfl: on lfs
                -i lflInit      -o ioerr        -c lflClose
                -r lflRead      -g lflGetc      -p lflPutc
                -w lflWrite     -s lflSeek      -n ioerr
                -intr NULL
```

**Figure 19.3** Configuration of types for a local file system master device and local file pseudo-device.

As the figure shows, driver functions for the master file system device have names that begin with *lfs*, and driver functions for a file pseudo-device have names that begin with *lfl*. We will see that support functions used by either set of driver functions have names that begin with *lf*.

## 19.13 The Local File System Open Function (lfsOpen)

Figure 19.4 shows the configuration for the master local file device and a set of local file pseudo-devices. Because a pseudo-device is used for each open file, the number of local file pseudo-devices provides a bound on the number of files that can be opened simultaneously.

```
/* Local file system master device (one per system)  */

            LFILESYS is lfs on disk

/* Local file pseudo-devices (many per system)       */

            LFILE0 is lfl on lfs
            LFILE1 is lfl on lfs
            LFILE2 is lfl on lfs
            LFILE3 is lfl on lfs
            LFILE4 is lfl on lfs
            LFILE5 is lfl on lfs
            LFILE6 is lfl on lfs
```

**Figure 19.4** Configuration of a local file system master device and a set of local file pseudo-devices.

The above configuration means that a process can use the master device to open a local file, and then use the pseudo-device to access the file. For example, to open a local file named *myfile* for reading and writing, a programmer codes:

```
fd = open(LFILESYS, "myfile", "rw");
```

Assuming the open succeeds, descriptor *fd* can be used to write data to the file, as in the following:

```
char buffer[1500];
... code to fill buffer ...
fd = write(fd, buffer, 1500);
```

Device *LFILESYS* is only used to open a file. Therefore, the master file system device driver only needs functions for *open* and *init*; all other I/O operations map to *ioerr*. Function *lfsOpen* performs the *open* operation; the code can be found in file *lfsOpen.c*.

```
/* lfsOpen.c  -  lfsOpen */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfsOpen - open a file and allocate a local file pseudo-device
 *------------------------------------------------------------------------
 */
devcall lfsOpen (
         struct dentry  *devptr,        /* entry in device switch table */
         char   *name,                  /* name of file to open         */
         char   *mode                   /* mode chars: 'r' 'w' 'o' 'n'  */
        )
```

```
{
        struct  lfdir   *dirptr;        /* ptr to in-memory directory   */
        char            *from, *to;     /* ptrs used during copy        */
        char            *nam, *cmp;     /* ptrs used during comparison  */
        int32           i;              /* general loop index           */
        did32           lfnext;         /* minor number of an unused    */
                                        /*    file pseudo-device        */
        struct  ldentry *ldptr;         /* ptr to an entry in directory */
        struct  lflcblk *lfptr;         /* ptr to open file table entry */
        bool8           found;          /* was the name found?          */
        int32   retval;                 /* value returned from function */
        int32   mbits;                  /* mode bits                    */

        /* Check length of name file (leaving space for NULLCH */

        from = name;
        for (i=0; i< LF_NAME_LEN; i++) {
                if (*from++ == NULLCH) {
                        break;
                }
        }
        if (i >= LF_NAME_LEN) {         /* name is too long */
                return SYSERR;
        }

        /* Parse mode argument and convert to binary */

        mbits = lfgetmode(mode);
        if (mbits == SYSERR) {
                return SYSERR;
        }

        /* If named file is already open, return SYSERR */

        lfnext = SYSERR;
        for (i=0; i<Nlfl; i++) {        /* search file pseudo-devices   */
                lfptr = &lfltab[i];
                if (lfptr->lfstate == LF_FREE) {
                        if (lfnext == SYSERR) {
                                lfnext = i; /* record index */
                        }
                        continue;
                }

                /* Compare requested name to name of open file */
```

```
        nam = name;
        cmp = lfptr->lfname;
        while(*nam != NULLCH) {
                if (*nam != *cmp) {
                        break;
                }
                nam++;
                cmp++;
        }

        /* See if comparison succeeded */

        if ( (*nam==NULLCH) && (*cmp == NULLCH) ) {
                return SYSERR;
        }
}
if (lfnext == SYSERR) { /* no slave file devices are available  */
        return SYSERR;
}

/* Obtain copy of directory if not already present in memory    */

dirptr = &Lf_data.lf_dir;
wait(Lf_data.lf_mutex);
if (! Lf_data.lf_dirpresent) {
    retval = read(Lf_data.lf_dskdev,(char *)dirptr,LF_AREA_DIR);
    if (retval == SYSERR ) {
        signal(Lf_data.lf_mutex);
        return SYSERR;
    }
    Lf_data.lf_dirpresent = TRUE;
}

/* Search directory to see if file exists */

found = FALSE;
for (i=0; i<dirptr->lfd_nfiles; i++) {
        ldptr = &dirptr->lfd_files[i];
        nam = name;
        cmp = ldptr->ld_name;
        while(*nam != NULLCH) {
                if (*nam != *cmp) {
                        break;
                }
```

```
                nam++;
                cmp++;
        }
        if ( (*nam==NULLCH) && (*cmp==NULLCH) ) { /* name found */
                found = TRUE;
                break;
        }
}

/* Case #1 - file is not in directory (i.e., does not exist)    */

if (! found) {
        if (mbits & LF_MODE_O) {        /* file *must* exist    */
                signal(Lf_data.lf_mutex);
                return SYSERR;
        }

        /* Take steps to create new file and add to directory   */

        /* Verify that space remains in the directory */

        if (dirptr->lfd_nfiles >= LF_NUM_DIR_ENT) {
                signal(Lf_data.lf_mutex);
                return SYSERR;
        }

        /* Allocate next dir. entry & initialize to empty file  */

        ldptr = &dirptr->lfd_files[dirptr->lfd_nfiles++];
        ldptr->ld_size = 0;
        from = name;
        to = ldptr->ld_name;
        while ( (*to++ = *from++) != NULLCH ) {
                ;
        }
        ldptr->ld_ilist = LF_INULL;

/* Case #2 - file is in directory (i.e., already exists)        */

} else if (mbits & LF_MODE_N) {         /* file must not exist  */
                signal(Lf_data.lf_mutex);
                return SYSERR;
}

/* Initialize the local file pseudo-device */
```

```
        lfptr = &lfltab[lfnext];
        lfptr->lfstate = LF_USED;
        lfptr->lfdirptr = ldptr;        /* point to directory entry     */
        lfptr->lfmode = mbits & LF_MODE_RW;

        /* File starts at position 0 */

        lfptr->lfpos     = 0;

        to = lfptr->lfname;
        from = name;
        while ( (*to = *from++) != NULLCH ) {
                ;
        }

        /* Neither index block nor data block are initially valid       */

        lfptr->lfinum    = LF_INULL;
        lfptr->lfdnum    = LF_DNULL;

        /* Initialize byte pointer to address beyond the end of the     */
        /*      buffer (i.e., invalid pointer triggers setup)           */

        lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ];
        lfptr->lfibdirty = FALSE;
        lfptr->lfdbdirty = FALSE;

        signal(Lf_data.lf_mutex);

        return lfptr->lfdev;
}
```

After verifying that the length of the file name is valid, *lfsOpen* calls *lfgetmode* to parse the *mode* argument and convert it to a set of bits. The mode argument consists of a null-terminated string that contains zero or more of the characters from Figure 19.5.

| Character | Meaning |
|---|---|
| r | Open the file for reading |
| w | Open the file for writing |
| o | File must be "old" (i.e., must exist) |
| n | File must be "new" (i.e., must not exist) |

**Figure 19.5** Characters permitted in a mode string and their meaning.

Characters in the mode string may not be repeated, and the combination of "o" and "n" is considered illegal. Furthermore, if neither "r" nor "w" is present, *lfgetmode* allows a default of both reading and writing. File *lfgetmode.c* contains the code.

```
/* lfgetmode.c  -  lfgetmode */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfgetmode  -  parse mode argument and generate integer of mode bits
 *------------------------------------------------------------------------
 */
int32   lfgetmode (
         char   *mode                   /* string of mode characters    */
        )
{
        int32   mbits;                  /* mode bits to return          */
        char    ch;                     /* next char in mode string     */

        mbits = 0;
        while ( (ch = *mode++) != NULLCH) {
                switch (ch) {

                    case 'r':   if (mbits&LF_MODE_R) {
                                        return SYSERR;
                                }
                                mbits |= LF_MODE_R;
                                continue;

                    case 'w':   if (mbits&LF_MODE_W) {
                                        return SYSERR;
```

```
                                }
                                mbits |= LF_MODE_W;
                                continue;

                        case 'o':   if (mbits&LF_MODE_O || mbits&LF_MODE_N) {
                                        return SYSERR;
                                }
                                mbits |= LF_MODE_O;
                                break;

                        case 'n':   if (mbits&LF_MODE_O || mbits&LF_MODE_N) {
                                        return SYSERR;
                                }
                                mbits |= LF_MODE_N;
                                break;

                        default:    return SYSERR;
                    }
            }

            /* If neither read nor write specified, allow both */

            if ( (mbits&LF_MODE_RW) == 0 ) {
                    mbits |= LF_MODE_RW;
            }
            return mbits;
}
```

Once the mode argument has been parsed, *lfsOpen* verifies that the file is not already open, verifies that a file pseudo-device is available, and searches the file system directory to see if the file exists. If the file exists (and the mode allows opening an existing file), *lfsOpen* fills in the control block of a file pseudo-device. If the file does not exist (and the mode allows creating a new file), *lfsOpen* allocates an entry in the file system directory, and then fills in the control block of a file pseudo-device. The initial file position is set to zero. Control block fields *lfinum* and *lfdnum* are set to the appropriate null value to indicate that neither the index block nor the data block are currently used. More important, field *lfbyte* is set to a value beyond the end of the data block buffer. We will see that setting *lfbyte* is important because the code uses the following invariant when accessing data:

> *When* lfbyte *contains an address in* lfdblock, *the byte to which it points contains the data that is in the file at the position given by* lfpos*; when* lfbyte *contains an address beyond* lfdblock, *values in* lfdblock *cannot be used*.

The details will become clear when we examine data transfer functions, such as *lflGetc* and *lflPutc*.

## 19.14 Closing A File Pseudo-Device (lflClose)

When an application finishes using a file, the application calls *close* to terminate use and make the pseudo-device available for other uses. In theory, closing a pseudo-device is trivial: change the state to indicate that the device is no longer in use. In practice, however, caching complicates closing because the control block may contain data that has not been written to the file. Thus, function *lflClose* must check bits in the control block that specify whether the index block or data block have been changed since they were written to disk. If changes have occurred, *lflClose* calls function *lfflush* to write the items to disk before it changes the state of the control block. File *lflClose.c* contains the code.

```
/* lflClose.c  -  lflClose.c */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflClose  --  close a file by flushing output and freeing device entry
 *------------------------------------------------------------------------
 */
devcall lflClose (
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  lflcblk *lfptr;         /* ptr to open file table entry */

        /* Obtain exclusive use of the file */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);

        /* If file is not open, return an error */

        if (lfptr->lfstate != LF_USED) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* Write index or data blocks to disk if they have changed */
```

```
        if (lfptr->lfdbdirty || lfptr->lfibdirty) {
                lfflush(lfptr);
        }

        /* Set device state to FREE and return to caller */

        lfptr->lfstate = LF_FREE;
        signal(lfptr->lfmutex);
        return OK;
}
```

## 19.15 Flushing Data To Disk (lfflush)

Function *lfflush* operates as expected. It receives a pointer to the control block of the pseudo-device as an argument, and uses the pointer to examine "dirty" bits in the control block†. If the index block has changed, *lfflush* uses *lfibput* to write a copy to disk; if the data block has changed, *lfflush* uses *write* to write a copy to disk. Fields *lfinum* and *lfdnum* contain the index block number and data block number to use. The code can be found in file *lfflush.c*.

```
/* lfflush.c  -  lfflush */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfflush  -  flush data block and index blocks for an open file
 *                      (assumes file mutex is held)
 *------------------------------------------------------------------------
 */
status  lfflush (
          struct lflcblk  *lfptr        /* ptr to file pseudo device    */
        )
{
        if (lfptr->lfstate == LF_FREE) {
                return SYSERR;
        }

        /* Write data block if it has changed */

        if (lfptr->lfdbdirty) {
                write(Lf_data.lf_dskdev, lfptr->lfdblock, lfptr->lfdnum);
                lfptr->lfdbdirty = FALSE;
        }
```

†The term *dirty bit* refers to a Boolean (i.e., a single bit) that is set to *TRUE* to indicate that data has changed.

```
        /* Write i-block if it has changed */

        if (lfptr->lfibdirty) {
                lfibput(Lf_data.lf_dskdev, lfptr->lfinum, &lfptr->lfiblock);
                lfptr->lfibdirty = FALSE;
        }

        return OK;
}
```

## 19.16 Bulk Transfer Functions For A File (lflWrite, lflRead)

Our implementation adopts a straightforward approach to writing and reading files: a loop that uses the appropriate character transfer function. For example, function *lflWrite*, which implements the *write* operation, calls *lflPutc* repeatedly. File *lflWrite.c* contains the code.

```
/* lflWrite.c  -  lfWrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflWrite  --  write data to a previously opened local disk file
 *------------------------------------------------------------------------
 */
devcall lflWrite (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer holding data to write */
          int32 count                   /* number of bytes to write     */
        )
{
        int32           i;              /* number of bytes written      */

        if (count < 0) {
                return SYSERR;
        }
        for (i=0; i<count; i++) {
                if (lflPutc(devptr, *buff++) == SYSERR) {
                        return SYSERR;
                }
        }
        return count;
}
```

Function *lflRead* implements the *read* operation. To satisfy a request, *lflRead* repeatedly calls *lflGetc*, receives a byte for each call, and places the byte in the next location of the caller's buffer. An interesting part of the code concerns how *lflRead* handles an *end-of-file* condition. When it reaches the end of the file, *lflGetc* returns constant *EOF*. If *lflRead* has already extracted one or more bytes of data from the file when it receives an *EOF* from *lflGetc*, *lflRead* stops the loop, and returns a count of the bytes that have been read. If no data has been found when an end-of-file is received, *lflRead* returns constant *EOF* to its caller. File *lflRead.c* contains the code.

```
/* lflRead.c  -  lfRead */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflRead  --  read from a previously opened local file
 *------------------------------------------------------------------------
 */
devcall lflRead (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer to hold bytes         */
          int32 count                   /* max bytes to read            */
        )
{
        uint32  numread;                /* number of bytes read         */
        int32   nxtbyte;                /* character or SYSERR/EOF      */

        if (count < 0) {
                return SYSERR;
        }

        for (numread=0 ; numread < count ; numread++) {
                nxtbyte = lflGetc(devptr);
                if (nxtbyte == SYSERR) {
                        return SYSERR;
                } else if (nxtbyte == EOF) {    /* EOF before finished */
                    if (numread == 0) {
                        return EOF;
                    } else {
                        return numread;
                    }
                } else {
                        *buff++ = (char) (0xff & nxtbyte);
                }
        }
```

```
        return numread;
}
```

## 19.17 Seeking To A New Position In the File (lflSeek)

A process can call *seek* to change the current position in a file. Our system uses function *lflSeek* to implement *seek*, and restricts the position to a valid point in the file (i.e., it is unlike Unix, which allows an application to seek beyond the end of the file).

Seeking to a new position consists of changing field *lfpos* in the file control block and setting field *lfbyte* to an address beyond *lfdblock* (which, according to the invariant above, means that the pointer cannot be used to extract data until the index block and data block are in place). File *lflSeek.c* contains the code.

```
/* lflSeek.c  -  lfSeek */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfseek  -  seek to a specified position in a file
 *------------------------------------------------------------------------
 */
devcall lflSeek (
          struct dentry *devptr,        /* entry in device switch table */
          uint32        offset          /* byte position in the file    */
        )
{
        struct  lflcblk *lfptr;         /* ptr to open file table entry */

        /* If file is not open, return an error */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);
        if (lfptr->lfstate != LF_USED) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* Verify offset is within current file size */

        if (offset > lfptr->lfdirptr->ld_size) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }
```

```
        /* Record new offset and invalidate byte pointer (i.e., */
        /*   force the index and data blocks to be replaced if  */
        /*   an attempt is made to read or write)               */

        lfptr->lfpos = offset;
        lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ];

        signal(lfptr->lfmutex);
        return OK;
}
```

## 19.18 Extracting One Byte From A File (lflGetc)

Once a file has been opened and both the correct index block and data block have been loaded into memory, extracting a byte from the file is trivial: it consists of treating *lfbyte* as a pointer to the byte, extracting the byte, and advancing the buffer pointer to the next byte. Function *lflGetc*, which performs the operation, can be found in file *lflGetc.c*.

```
/* lflGetc.c  -  lfGetc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflGetc  --  Read the next byte from an open local file
 *------------------------------------------------------------------------
 */
devcall lflGetc (
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  lflcblk *lfptr;         /* ptr to open file table entry */
        struct  ldentry *ldptr;         /* ptr to file's entry in the   */
                                        /*  in-memory directory         */
        int32   onebyte;                /* next data byte in the file   */

        /* Obtain exclusive use of the file */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);
```

```
        /* If file is not open, return an error */

        if (lfptr->lfstate != LF_USED) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* Return EOF for any attempt to read beyond the end-of-file */

        ldptr = lfptr->lfdirptr;
        if (lfptr->lfpos >= ldptr->ld_size) {
                signal(lfptr->lfmutex);
                return EOF;
        }

        /* If byte pointer is beyond the current data block, */
        /*      set up a new data block                      */

        if (lfptr->lfbyte >= &lfptr->lfdblock[LF_BLKSIZ]) {
                lfsetup(lfptr);
        }

        /* Extract the next byte from block, update file position, and  */
        /*      return the byte to the caller                           */

        onebyte = 0xff & *lfptr->lfbyte++;
        lfptr->lfpos++;
        signal(lfptr->lfmutex);
        return onebyte;
}
```

If the file is not open, *lflGetc* returns *SYSERR*. If the current file position exceeds the file size, *lflGetc* returns *EOF*. In other cases, *lflGetc* checks pointer *lfbyte* to see whether it currently points outside of the data block in *lfdblock*. If so, *lflGetc* calls *lfsetup* to read the correct index block and data block into memory.

Once the data block is in memory, *lflGetc* can extract a byte. To do so, it dereferences *lfbyte* to obtain a byte and places the byte in variable *onebyte*. It increments both the byte pointer and the file position before returning the byte to the caller.

## 19.19 Changing One Byte In A File (lflPutc)

Function *lflPutc* handles the task of storing a byte into a file at the current position. As with *lflGetc*, the code that performs the transfer is trivial and only occupies a few lines. Pointer *lfbyte* gives a location in *lfdblock* at which the byte should be stored; the code uses the pointer, stores the specified byte, increments the pointer, and sets *lfdbdirty* to indicate that the data block has been changed. Note that *lflPutc* merely accumulates characters in a buffer in memory; it does not write the buffer to disk each time a change occurs. The buffer will only be copied to disk when the current position moves to another disk block.

Like *lflGetc*, *lflPutc* examines *lfbyte* on each call. If *lfbyte* lies outside of data block *lfdblock*, *lflPutc* returns *SYSERR*. However, a subtle difference exists in the way *lflPutc* and *lflGetc* treat an invalid file position. *LflGetc* always returns *EOF* if the file position exceeds the last byte of the file. *LflPutc* returns *SYSERR* when the file position is more than one byte beyond the end of the file, but if the position is exactly one byte beyond the end, it allows the operation to proceed. That is, it allows a file to be extended. When a file is extended, the file size, found in the directory entry for the file, must be incremented. File *lflPutc.c* contains the code.

```
/* lflPutc.c  -  lfPutc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflPutc  -  write a single byte to an open local file
 *------------------------------------------------------------------------
 */
devcall lflPutc (
          struct dentry *devptr,        /* entry in device switch table */
          char          ch              /* character (byte) to write    */
        )
{
        struct  lflcblk *lfptr;         /* ptr to open file table entry */
        struct  ldentry *ldptr;         /* ptr to file's entry in the   */
                                        /*  in-memory directory         */

        /* Obtain exclusive use of the file */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);

        /* If file is not open, return an error */

        if (lfptr->lfstate != LF_USED) {
```

```
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* Return SYSERR for an attempt to skip bytes beyond the */
        /*      current end of the file                          */

        ldptr = lfptr->lfdirptr;
        if (lfptr->lfpos > ldptr->ld_size) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* If pointer is outside current block, set up new block */

        if (lfptr->lfbyte >= &lfptr->lfdblock[LF_BLKSIZ]) {

                /* Set up block for current file position */
                lfsetup(lfptr);
        }

        /* If appending a byte to the file, increment the file size.    */
        /* Note: comparison might be equal, but should not be greater.  */

        if (lfptr->lfpos >= ldptr->ld_size) {
                ldptr->ld_size++;
        }

        /* Place byte in buffer and mark buffer "dirty" */

        *lfptr->lfbyte++ = ch;
        lfptr->lfpos++;
        lfptr->lfdbdirty = TRUE;

        signal(lfptr->lfmutex);
        return OK;
}
```

## 19.20 Loading An Index Block And A Data Block (lfsetup)

Once a file position has been assigned to field *lfpos*, function *lfsetup* loads a copy of the index block and data block associated with the position from the disk. *Lfsetup* begins by obtaining pointers to data structures. If the existing index or data blocks have

changed, *lfsetup* calls *lfflush* to write them to disk. It then examines the index block in the file control block.

The first step in loading data for the current file position consists of loading an index block that either precedes or coincides with the current position. There are two cases. If no index block has been loaded (i.e., the file was just opened), *lfsetup* obtains one. For a new file, *lfsetup* must allocate an initial index block from the free list; for an existing file, it loads the first index block for the file. If an index block has already been loaded, *lfsetup* must handle the case where the block corresponds to a portion of the file that lies after the current file position (e.g., a process has issued a *seek* to an earlier position). To handle the case, *lfsetup* replaces the index block with the initial index block for the file.

Once it has loaded an index block, *lfsetup* enters a loop that moves along the linked list of index blocks until it reaches the index block that covers the current file position. At each iteration, *lfsetup* uses field *ib_next* to find the number of the next index block on the list, and then calls *lfibget* to read the index block into memory.

Once the correct index block has been loaded, *lfsetup* must determine the data block to load. To do so, it uses the file position to compute an index (from 0 through 15) for the data block array. Because each index block covers exactly 8K bytes (i.e., $2^{13}$ bytes) of data and each slot in the array corresponds to a block of 512 bytes ($2^9$), binary arithmetic can be used. *Lfsetup* computes the *logical and* of the *LF_IMASK* (the low-order 13 bits) and then shifts the result right 9 bits.

*Lfsetup* uses the result of the above computation as an index into array *ib_dba* to obtain the ID of a data block. There are two cases that require *lfsetup* to load a new data block. In the first case, the pointer in the array is null, which means *lflPutc* is about to write a new byte on the end of the file and no data block has been assigned for the position. *Lfsetup* calls *lfdballoc* to allocate a new data block from the free list, and records the ID in the entry of array *ib_dba*. In the second case, the entry in array *ib_dba* specifies a data block other than the data block currently loaded. *Lfsetup* calls *read* to fetch the correct data block from disk.

As the final step before returning, *lfsetup* uses the file position to compute a position within the data block, and assigns the address to field *lfbyte*. The careful arrangement of making the data block size a power of two means that the indices from 0 through 511 can be computed by selecting the low-order 9 bits of the file position. The code uses a *logical and* with mask *LF_DMASK*. File *lfsetup.c* contains the code.

```
/* lfsetup.c  -  lfsetup */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfsetup  -  set a file's index block and data block for the current
 *               file position (assumes file mutex held)
 *------------------------------------------------------------------------
 */
```

```
status  lfsetup (
          struct lflcblk  *lfptr        /* ptr to slave file device     */
        )
{
        dbid32  dnum;                   /* data block to fetch          */
        ibid32  ibnum;                  /* i-block number during search */
        struct  ldentry *ldptr;         /* ptr to file entry in dir.    */
        struct  lfiblk  *ibptr;         /* ptr to in-memory index block */
        uint32  newoffset;              /* computed data offset for     */
                                        /*  next index block            */
        int32   dindex;                 /* index into array in an index */
                                        /*  block                       */


        /* Obtain exclusive access to the directory */

        wait(Lf_data.lf_mutex);

        /* Get pointers to in-memory directory, file's entry in the     */
        /*      directory, and the in-memory index block                */

        ldptr = lfptr->lfdirptr;
        ibptr = &lfptr->lfiblock;

        /* If existing index block or data block changed, write to disk */

        if (lfptr->lfibdirty || lfptr->lfdbdirty) {
                lfflush(lfptr);
        }
        ibnum = lfptr->lfinum;          /* get ID of curr. index block  */

        /* If there is no index block in memory (e.g., because the file */
        /*      was just opened), either load the first index block of  */
        /*      the file or allocate a new first index block            */

        if (ibnum == LF_INULL) {

                /* Check directory entry to see if index block exists   */

                ibnum = ldptr->ld_ilist;
                if (ibnum == LF_INULL) { /* empty file - get new i-block*/
                        ibnum = lfiballoc();
                        lfibclear(ibptr, 0);
                        ldptr->ld_ilist = ibnum;
                        lfptr->lfibdirty = TRUE;
```

```
        } else {                /* nonempty - read first i-block*/
                lfibget(Lf_data.lf_dskdev, ibnum, ibptr);
        }
        lfptr->lfinum = ibnum;

/* Otherwise, if current file position has been moved to an     */
/*   offset before the current index block, start at the        */
/*   beginning of the index list for the file                   */

} else if (lfptr->lfpos < ibptr->ib_offset) {

        /* Load initial index block for the file (we know that  */
        /*      at least one index block exists)                */

        ibnum = ldptr->ld_ilist;
        lfibget(Lf_data.lf_dskdev, ibnum, ibptr);
        lfptr->lfinum = ibnum;
}

/* At this point, an index block is in memory, but may cover    */
/*  an offset less than the current file position.  Loop until  */
/*  the index block covers the current file position.           */

while ((lfptr->lfpos & ~LF_IMASK) > ibptr->ib_offset ) {
        ibnum = ibptr->ib_next;
        if (ibnum == LF_INULL) {
                /* allocate new index block to extend file */
                ibnum = lfiballoc();
                ibptr->ib_next = ibnum;
                lfibput(Lf_data.lf_dskdev, lfptr->lfinum, ibptr);
                lfptr->lfinum = ibnum;
                newoffset = ibptr->ib_offset + LF_IDATA;
                lfibclear(ibptr, newoffset);
                lfptr->lfibdirty = TRUE;
        } else {
                lfibget(Lf_data.lf_dskdev, ibnum, ibptr);
                lfptr->lfinum = ibnum;
        }
        lfptr->lfdnum = LF_DNULL; /* Invalidate old data block */
}
```

```
        /* At this point, the index block in lfiblock covers the        */
        /*  current file position (i.e., position lfptr->lfpos).  The   */
        /*  next step consists of loading the correct data block.       */

        dindex = (lfptr->lfpos & LF_IMASK) >> 9;

        /* If data block index does not match current data block, read  */
        /*   the correct data block from disk                           */

        dnum = lfptr->lfiblock.ib_dba[dindex];
        if (dnum == LF_DNULL) {         /* allocate new data block */
                dnum = lfdballoc((struct lfdbfree *)&lfptr->lfdblock);
                lfptr->lfiblock.ib_dba[dindex] = dnum;
                lfptr->lfibdirty = TRUE;
        } else if ( dnum != lfptr->lfdnum) {
                read(Lf_data.lf_dskdev, (char *)lfptr->lfdblock, dnum);
                lfptr->lfdbdirty = FALSE;
        }
        lfptr->lfdnum = dnum;

        /* Use current file offset to set the pointer to the next byte  */
        /*   within the data block                                      */

        lfptr->lfbyte = &lfptr->lfdblock[lfptr->lfpos & LF_DMASK];
        signal(Lf_data.lf_mutex);
        return OK;
}
```

## 19.21 Master File System Device Initialization (lfsInit)

Initialization for the master file system device is straightforward. Function *lfsInit* performs the task, which consists of recording the ID of the disk device, creating a semaphore that provides mutual exclusion to the directory, clearing the in-memory directory (merely to help with debugging), and setting a Boolean to indicate that the directory has not been read into memory. Data for the master file system device is kept in the global structure *Lf_data*. File *lfsInit.c* contains the code.

```
/* lfsInit.c  -  lfsInit */

#include <xinu.h>

struct  lfdata  Lf_data;

/*------------------------------------------------------------------------
 * lfsInit  --  initialize the local file system master device
 *------------------------------------------------------------------------
 */
devcall lfsInit (
          struct dentry *devptr         /* entry in device switch table */
        )
{
        /* Assign ID of disk device that will be used */

        Lf_data.lf_dskdev = LF_DISK_DEV;

        /* Create a mutual exclusion semaphore */

        Lf_data.lf_mutex = semcreate(1);

        /* Zero directory area (for debugging) */

        memset((char *)&Lf_data.lf_dir, NULLCH, sizeof(struct lfdir));

        /* Initialize directory to "not present" in memory */

        Lf_data.lf_dirpresent = Lf_data.lf_dirdirty = FALSE;

        return OK;
}
```

## 19.22 Pseudo-Device Initialization (lflInit)

When it opens a file, *lfsOpen* initializes many of the entries in the control block for the file pseudo-device. However, some initialization is performed at system startup. To indicate that the device is not in use, the state is assigned *LF_FREE*. A mutual exclusion semaphore is created to guarantee that at most one operation will be in progress on the file at a given time. Most other fields in the control block are initialized to zero (they will not be used until a file is opened, but initializing to zero can make debugging easier). File *lflInit.c* contains the code.

```
/* lflInit.c  -  lflInit */

#include <xinu.h>

struct  lflcblk lfltab[Nlfl];           /* control blocks */

/*------------------------------------------------------------------------
 * lflInit  -  initialize control blocks for local file pseudo-devices
 *------------------------------------------------------------------------
 */
devcall lflInit (
          struct dentry *devptr         /* Entry in device switch table */
                )
{
        struct  lflcblk *lfptr;         /* Ptr. to control block entry  */
        int32   i;                      /* Walks through name array     */

        lfptr = &lfltab[ devptr->dvminor ];

        /* Initialize control block entry */

        lfptr->lfstate = LF_FREE;       /* Device is currently unused   */
        lfptr->lfdev = devptr->dvnum;   /* Set device ID                */
        lfptr->lfmutex = semcreate(1);
        lfptr->lfdirptr = (struct  ldentry *) NULL;
        lfptr->lfpos = 0;
        for (i=0; i<LF_NAME_LEN; i++) {
                lfptr->lfname[i] = NULLCH;
        }
        lfptr->lfinum = LF_INULL;
        memset((char *) &lfptr->lfiblock, NULLCH, sizeof(struct lfiblk));
        lfptr->lfdnum = 0;
        memset((char *) &lfptr->lfdblock, NULLCH, LF_BLKSIZ);
        lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ]; /* beyond lfdblock */
        lfptr->lfibdirty = lfptr->lfdbdirty = FALSE;
        return OK;
}
```

## 19.23 File Truncation (lftruncate)

We will use file truncation as a way to show how file data structures are deallocated. To truncate a file to zero length, each of the index blocks for the file must be placed on the free list of index blocks. Before an index block can be released, however, each of the data blocks to which the index block points must be placed on the free list

of data blocks. Function *lftruncate* performs file truncation; file *lftruncate.c* contains the code.

```
/* lftruncate.c  -  lftruncate */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lftruncate  -  truncate a file by freeing its index and data blocks
 *                      (assumes directory mutex held)
 *------------------------------------------------------------------------
 */
status  lftruncate (
          struct lflcblk *lfptr         /* ptr to file's cntl blk entry */
        )
{
        struct  ldentry *ldptr;         /* pointer to file's dir. entry */
        struct  lfiblk  iblock;         /* buffer for one index block   */
        ibid32  ifree;                  /* start of index blk free list */
        ibid32  firstib;                /* first index blk of the file  */
        ibid32  nextib;                 /* walks down list of the       */
                                        /*   file's index blocks        */
        dbid32  nextdb;                 /* next data block to free      */
        int32   i;                      /* moves through data blocks in */
                                        /*   a given index block        */

        ldptr = lfptr->lfdirptr;        /* Get pointer to dir. entry    */
        if (ldptr->ld_size == 0) {      /* file is already empty */
                return OK;
        }

        /* Clean up the open local file first */

        if ( (lfptr->lfibdirty) || (lfptr->lfdbdirty) ) {
                lfflush(lfptr);
        }
        lfptr->lfpos = 0;
        lfptr->lfinum = LF_INULL;
        lfptr->lfdnum = LF_DNULL;
        lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ];

        /* Obtain ID of first index block on free list */

        ifree = Lf_data.lf_dir.lfd_ifree;
```

```
        /* Record file's first i-block and clear directory entry */

        firstib = ldptr->ld_ilist;
        ldptr->ld_ilist = LF_INULL;
        ldptr->ld_size = 0;
        Lf_data.lf_dirdirty = TRUE;

        /* Walk along index block list, disposing of each data block    */
        /*  and clearing the corresponding pointer.  A note on loop     */
        /*  termination: last pointer is set to ifree below.            */

        for (nextib=firstib; nextib!=ifree; nextib=iblock.ib_next) {

                /* Obtain a copy of current index block from disk       */

                lfibget(Lf_data.lf_dskdev, nextib, &iblock);

                /* Free each data block in the index block              */

                for (i=0; i<LF_IBLEN; i++) {    /* for each d-block     */

                        /* Free the data block */

                        nextdb = iblock.ib_dba[i];
                        if (nextdb != LF_DNULL) {
                                lfdbfree(Lf_data.lf_dskdev, nextdb);
                        }

                        /* Clear entry in i-block for this d-block      */

                                iblock.ib_dba[i] = LF_DNULL;
                }

                /* Clear offset (just to make debugging easier)         */

                iblock.ib_offset = 0;

                /* For the last index block on the list, make it point  */
                /*      to the current free list                        */

                if (iblock.ib_next == LF_INULL) {
                        iblock.ib_next = ifree;
                }

                /* Write cleared i-block back to disk */
```

```
                lfibput(Lf_data.lf_dskdev, nextib, &iblock);
        }

        /* Last index block on the file list now points to first node   */
        /*   on the current free list.  Once we make the free list      */
        /*   point to the first index block on the file list, the       */
        /*   entire set of index blocks will be on the free list        */

        Lf_data.lf_dir.lfd_ifree = firstib;

        /* Indicate that directory has changed and return */

        Lf_data.lf_dirdirty = TRUE;

        return OK;
}
```

The approach used is straightforward: if the file already has length zero, return to the caller. Otherwise, walk along the file's index block list. Read each index block into memory, and call *lfdbfree* to free each data block to which the index block points.

Once the final index block has been reached, add all the index blocks for the file to the free list. To do so, observe that all index blocks for the file are already linked. Thus, only two pointer changes are needed. First, change the *next* pointer in the final index block of the file to point to the current free list. Second, change the free list to point to the first index block of the file.

## 19.24 Initial File System Creation (lfscreate)

A final initialization function will complete details of the file system. Function *lfscreate* creates an initial, empty file system on a disk. That is, it forms a free list of index blocks, a free list of data blocks, and a directory that contains no files. File *lfscreate.c* contains the code.

```
/* lfscreate.c  -  lfscreate */

#include <xinu.h>
#include <ramdisk.h>

/*------------------------------------------------------------------------
 * lfscreate  --  Create an initially-empty file system on a disk
 *------------------------------------------------------------------------
 */
```

```
status  lfscreate (
          did32         disk,           /* ID of an open disk device    */
          ibid32        lfiblks,        /* num. of index blocks on disk */
          uint32        dsiz            /* total size of disk in bytes  */
        )
{
        uint32  sectors;                /* number of sectors to use     */
        uint32  ibsectors;              /* number of sectors of i-blocks*/
        uint32  ibpersector;            /* number of i-blocks per sector*/
        struct  lfdir   dir;            /* Buffer to hold the directory */
        uint32  dblks;                  /* total free data blocks       */
        struct  lfiblk  iblock;         /* space for one i-block        */
        struct  lfdbfree dblock;        /* data block on the free list  */
        dbid32  dbindex;                /* index for data blocks        */
        int32   retval;                 /* return value from func call  */
        int32   i;                      /* loop index                   */

        /* Compute total sectors on disk */

        sectors = dsiz  / LF_BLKSIZ;    /* truncate to full sector */

        /* Compute number of sectors comprising i-blocks */

        ibpersector = LF_BLKSIZ / sizeof(struct lfiblk);
        ibsectors = (lfiblks+(ibpersector-1)) / ibpersector; /* round up*/
        lfiblks = ibsectors * ibpersector;
        if (ibsectors > sectors/2) {    /* invalid arguments */
                return SYSERR;
        }

        /* Create an initial directory */

        memset((char *)&dir, NULLCH, sizeof(struct lfdir));
        dir.lfd_nfiles = 0;
        dbindex= (dbid32)(ibsectors + 1);
        dir.lfd_dfree = dbindex;
        dblks = sectors - ibsectors - 1;
        retval = write(disk,(char *)&dir, LF_AREA_DIR);
        if (retval == SYSERR) {
                return SYSERR;
        }

        /* Create list of free i-blocks on disk */

        lfibclear(&iblock, 0);
```

```
        for (i=0; i<lfiblks-1; i++) {
                iblock.ib_next = (ibid32)(i + 1);
                lfibput(disk, i, &iblock);
        }
        iblock.ib_next = LF_INULL;
        lfibput(disk, i, &iblock);

        /* Create list of free data blocks on disk */

        memset((char*)&dblock, NULLCH, LF_BLKSIZ);
        for (i=0; i<dblks-1; i++) {
                dblock.lf_nextdb = dbindex + 1;
                write(disk, (char *)&dblock, dbindex);
                dbindex++;
        }
        dblock.lf_nextdb = LF_DNULL;
        write(disk, (char *)&dblock, dbindex);
        close(disk);
        return OK;
}
```

## 19.25 Perspective

File systems are among the most complex pieces of an operating system. Our implementation avoids one of the most challenging problems, sharing, by imposing a restriction: a file can only be opened once at a given time. If the restriction is relaxed, a file system must coordinate operations among multiple file descriptors that can refer to the same file. Sharing raises the question of semantics: how should overlapping *write* operations be interpreted? In particular, if a process attempts to write bytes 0 through N of a file and another process simultaneously attempts to write bytes 2 through N-1 of the same file, what should happen? Should the file system guarantee that one of the two operations occurs first? Should the file system allow bytes to be intermingled? How should the file system manage a shared cache to make operations efficient?

A second form of complexity arises from the implementation. All operations on files must be translated into operations on disk blocks. As a result, basic data structures, such as linked lists, can be complex to manipulate. Interestingly, much of the complexity arises when disk blocks are shared. For example, because a given disk block can hold index blocks from multiple files, two processes may need to access the same disk block simultaneously. Most file systems arrange to cache disk blocks, making such access efficient.

A final form of complexity arises from the need for safety and recovery. Users assume that once data has been written to a file, the data is "safe" even if the power fails. However, a file system cannot afford to write to disk each time an application stores a byte in a file. Thus, one of the grand challenges of file system design arises from the

tradeoff between efficiency and safety — a designer looks for ways to minimize the risk of losing data while also looking for ways to maximize efficiency.

## 19.26 Summary

A file system manages objects on nonvolatile storage. To insure the interface to files is the same as the interface to devices, our example system is organized into a master file system device and a set of file pseudo-devices. To access a file, a process *opens* the master device; the call returns the descriptor of a pseudo-device for the file. Once a file has been opened, functions *read*, *write*, *getc*, *putc*, *seek* and *close* can be used on the file.

Our design allows files to grow dynamically; the data structures for a file consist of a directory entry and a linked list of index blocks that each point to a set of data blocks. When a file is used, the driver software loads an index block and data block into memory. Subsequent accesses or changes to a file affect the data block in memory. When the file position moves outside the current block, the file system writes the data block back to disk and allocates another data block. Similarly, when the file position moves outside the data covered by the current index block, the system writes the current index block to disk and allocates a new index block.

## EXERCISES

**19.1** Redesign routines *lflRead* and *lflWrite* to perform high-speed copies (i.e., copy bytes from or to the current data block without making repeated calls to *lflGetc* or *lflPutc*. Redesign the system to permit multiple processes to *open* the same file simultaneously. Coordinate all writes to insure that a given byte in the file always contains the data written last.

**19.2** Free data blocks are chained together on a singly-linked list. Redesign the system to place them in a file (i.e., reserve index block 0 to be an unnamed "file" in which index blocks point to free data blocks). Compare the performance of the new and original designs.

**19.3** What are the maximum number of disk accesses necessary to allocate and free a data block under the original design and the new design in the previous exercise?

**19.4** The number of index blocks is important because having too many wastes space that could be used for data, while having too few means data blocks will be wasted because there are insufficient index blocks to use them. Given that there are 16 data block pointers in an index block and 7 index blocks fill a disk block, how many index blocks might be needed for a disk of $n$ total blocks if the directory can hold $k$ files?

**19.5** Current index block IDs are 32 bits long. Redesign the system to use 16-bit index block IDs. What are the tradeoffs of the two approaches?

**19.6** Redesign the system so it closes all files that have been opened by a process when the process terminates.

**19.7** Change the system to have a file switch table separate from the device switch table. What are the advantages and disadvantages of each approach?

**19.8** After changing the free list, function *lfiballoc* writes a copy of the directory to disk. As an alternative, *lfiballoc* could mark the directory "dirty" and defer the *write* operation until later. Discuss the advantages and disadvantages of each approach.

**19.9** Consider two processes attempting to *write* to a single file. Suppose one process repeatedly writes 20 bytes of character *A* and the other process repeatedly writes 20 bytes of character *B*. Describe the order in which characters might appear in the file.

**19.10** Create a *control* function for the file pseudo-device driver that allows a caller to invoke *lftruncate*.

**19.11** Create a *control* function for the master file system device that allows a caller to invoke *lfscreate*.

# NOTES

# Chapter Contents

# 20

# A Remote File Mechanism

*Networking makes the far-away the here-and-now.*

— Unknown

## 20.1 Introduction

Chapter 16 discusses a network interface and a device driver that uses the hardware interface to send and receive packets. Chapter 18 considers disk hardware and the block transfer paradigm. Chapter 19 explains how a file system creates high-level abstractions, including dynamic files, and shows how files can be mapped onto a disk.

This chapter expands the discussion of file systems by considering an alternative that uses a remote file server. That is, instead of implementing the file abstraction directly on hardware, the operating system relies on a separate computer called a server. When an application requests a file operation, the operating system sends a request to the server and receives a response. The next chapter extends the discussion by showing how a remote and local file system can be integrated.

## 20.2 Remote File Access

A remote file access mechanism requires four conceptual pieces. First, an operating system must contain a device driver for a network device (such as an Ethernet). Second, the operating system must also contain protocol software (such as UDP and IP) that handles addressing so the packets can reach the remote server and replies can return. Third, the operating system must have remote file access software that becomes a client (i.e., forms a request, uses the network to send the request to the server and re-

ceive a response, and interprets the response). Whenever a process invokes an I/O operation on a remote file (e.g., *read* or *write*), the remote file access software forms a message that specifies the operation, sends the request to the remote file server, and processes the response. Fourth, a computer on the network must be running a remote file server application that honors each request.

In practice, many questions arise about the design of a remote file access mechanism. What services should a remote file server provide? Should the service permit a client to create hierarchical directories, or should the server only permit a client to create data files? Should the mechanism allow a client to remove files? If two or more clients send requests to a given server, should the files be shared or should each client have its own files? Should a file be cached in the memory of the client machine? For example, when a process reads a byte from a remote file, should the client software request one-thousand bytes and hold the extra bytes in a cache to avoid sending requests to the remote server for successive bytes?

## 20.3 Remote File Semantics

One of the primary design considerations surrounding remote file systems arises from heterogeneity: the operating system on the client and server machines may differ. As a result, the file operations available to the remote server may differ from the file operations used on the client machine. For example, because the remote file server used with Xinu runs on a Unix system (e.g., Linux or Solaris), the server supplies functionality from the Unix file system.

Most of the Xinu file operations map directly to Unix file operations. For example, Xinu uses the same semantics for *read* as Unix — a request specifies a buffer size and *read* specifies the number of data bytes that were placed in the buffer. Similarly, a Xinu *write* operation follows the same semantics as a Unix *write*.

However, Xinu semantics do differ from Unix semantics in many ways. Each Unix file has an owner that is identified by a Unix userid; Xinu does not have userids, and even if it did, they would not align with the userids used by the server. Even small details differ. For example, the *mode* argument used with a Xinu *open* operation allows a caller to specify that the file must be *new* (i.e., must not exist) or that the file must be *old* (i.e., must exist). Unix allows a file to be created, but does not test whether the file already exists. Instead, if the file exists, Unix truncates the file to zero bytes. Thus, to implement the Xinu *new* mode, a remote server running on a Unix system must first test whether the file exists and return an error indication if it does.

## 20.4 Remote File Design And Messages

Our example remote file system provides basic functionality: a Xinu process can create a file, write data to the file, seek to an arbitrary position in the file, read data from the file, truncate a file, and delete a file. In addition, the remote file system allows

a Xinu process to create or remove directories. For each operation, the system defines a request message (sent from a Xinu client to the remote file server) and a response message (sent from the server back to the Xinu client). Each message begins with a common header that specifies the type of the operation, a status value (used in responses to report errors), a sequence number, and the name of a file. Each outgoing request is assigned a unique sequence number, and the remote file software checks a reply to insure that an incoming reply matches the outgoing request. Our implementation defines a structure for each message type. To avoid nested structure declarations, the code uses a preprocessor definition, *RF_MSG_HDR*, for the header fields, and then includes the header in each struct. File *rfilesys.h* contains the code.

```
/* rfilesys.h - definitions for remote file system pseudo-devices */

#ifndef Nrfl
#define Nrfl    10
#endif

/* Control block for a remote file pseudo-device */

#define RF_NAMLEN       128             /* Maximum length of file name  */
#define RF_DATALEN      1024            /* Maximum data in read or write*/
#define RF_MODE_R       F_MODE_R        /* Bit to grant read access     */
#define RF_MODE_W       F_MODE_W        /* Bit to grant write access    */
#define RF_MODE_RW      F_MODE_RW       /* Mask for read and write bits */
#define RF_MODE_N       F_MODE_N        /* Bit for "new" mode           */
#define RF_MODE_O       F_MODE_O        /* Bit for "old" mode           */
#define RF_MODE_NO      F_MODE_NO       /* Mask for "n" and "o" bits    */

/* Global data for the remote server */

#ifndef RF_SERVER_IP
#define RF_SERVER_IP    "255.255.255.255"
#endif

#ifndef RF_SERVER_PORT
#define RF_SERVER_PORT  33123
#endif

#ifndef RF_LOC_PORT
#define RF_LOC_PORT     33123
#endif

struct  rfdata  {
        int32   rf_seq;                 /* next sequence number to use  */
        uint32  rf_ser_ip;              /* server IP address            */
```

```
        uint16  rf_ser_port;            /* server UDP port              */
        uint16  rf_loc_port;            /* local (client) UPD port      */
        sid32   rf_mutex;               /* mutual exclusion for access  */
        bool8   rf_registered;          /* has UDP port been registered?*/
};

extern  struct  rfdata  Rf_data;

/* Definition of the control block for a remote file pseudo-device      */

#define RF_FREE 0                       /* Entry is currently unused    */
#define RF_USED 1                       /* Entry is currently in use    */

struct  rflcblk {
        int32   rfstate;                /* entry is free or used        */
        int32   rfdev;                  /* device number of this dev.   */
        char    rfname[RF_NAMLEN];      /* Name of the file             */
        uint32  rfpos;                  /* current file position        */
        uint32  rfmode;                 /* mode: read access, write     */
                                        /*      access or both          */
};

extern  struct  rflcblk rfltab[];       /* remote file control blocks   */

/* Definitions of parameters used when accessing a remote server        */

#define RF_RETRIES      3               /* time to retry sending a msg  */
#define RF_TIMEOUT      1000            /* wait one second for a reply  */

/* Control functions for a remote file pseudo device */

#define RFS_CTL_DEL     F_CTL_DEL       /* Delete a file                */
#define RFS_CTL_TRUNC   F_CTL_TRUNC     /* Truncate a file              */
#define RFS_CTL_MKDIR   F_CTL_MKDIR     /* make a directory             */
#define RFS_CTL_RMDIR   F_CTL_RMDIR     /* remove a directory           */
#define RFS_CTL_SIZE    F_CTL_SIZE      /* Obtain the size of a file    */

/************************************************************************/
/*                                                                      */
/*      Definition of messages exchanged with the remote server         */
/*                                                                      */
/************************************************************************/

/* Values for the type field in messages */
```

```
#define RF_MSG_RESPONSE 0x0100          /* Bit that indicates response  */

#define RF_MSG_RREQ     0x0001          /* Read Request and response    */
#define RF_MSG_RRES     (RF_MSG_RREQ | RF_MSG_RESPONSE)

#define RF_MSG_WREQ     0x0002          /* Write Request and response   */
#define RF_MSG_WRES     (RF_MSG_WREQ | RF_MSG_RESPONSE)

#define RF_MSG_OREQ     0x0003          /* Open request and response    */
#define RF_MSG_ORES     (RF_MSG_OREQ | RF_MSG_RESPONSE)

#define RF_MSG_DREQ     0x0004          /* Delete request and response  */
#define RF_MSG_DRES     (RF_MSG_DREQ | RF_MSG_RESPONSE)

#define RF_MSG_TREQ     0x0005          /* Truncate request & response  */
#define RF_MSG_TRES     (RF_MSG_TREQ | RF_MSG_RESPONSE)

#define RF_MSG_SREQ     0x0006          /* Size request and response    */
#define RF_MSG_SRES     (RF_MSG_SREQ | RF_MSG_RESPONSE)

#define RF_MSG_MREQ     0x0007          /* Mkdir request and response   */
#define RF_MSG_MRES     (RF_MSG_MREQ | RF_MSG_RESPONSE)

#define RF_MSG_XREQ     0x0008          /* Rmdir request and response   */
#define RF_MSG_XRES     (RF_MSG_XREQ | RF_MSG_RESPONSE)

#define RF_MIN_REQ      RF_MSG_RREQ     /* Minimum request type         */
#define RF_MAX_REQ      RF_MSG_XREQ     /* Maximum request type         */

/* Message header fields present in each message */

#define RF_MSG_HDR                      /* Common message fields        */\
        uint16  rf_type;                /* message type                 */\
        uint16  rf_status;              /* 0 in req, status in response */\
        uint32  rf_seq;                 /* message sequence number      */\
        char    rf_name[RF_NAMLEN];     /* null-terminated file name    */

/* The standard header present in all messages with no extra fields     */

/************************************************************************/
/*                                                                      */
/*                              Header                                  */
/*                                                                      */
/************************************************************************/
```

```
#pragma pack(2)
struct  rf_msg_hdr {                    /* header fields present in each*/
        RF_MSG_HDR                      /*   remote file system message */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Read                                    */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_rreq     {               /* remote file read request     */
        RF_MSG_HDR                      /* header fields                */
        uint32  rf_pos;                 /* position in file to read     */
        uint32  rf_len;                 /* number of bytes to read      */
                                        /*   (between 1 and 1024)       */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_rres     {               /* remote file read reply       */
        RF_MSG_HDR                      /* header fields                */
        uint32  rf_pos;                 /* position in file             */
        uint32  rf_len;                 /* number of bytes that follow  */
                                        /*   (0 for EOF)                */
        char    rf_data[RF_DATALEN];    /* array containing data from   */
                                        /*   the file                   */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Write                                   */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_wreq     {               /* remote file write request    */
        RF_MSG_HDR                      /* header fields                */
        uint32  rf_pos;                 /* position in file             */
        uint32  rf_len;                 /* number of valid bytes in     */
                                        /*   array that follows         */
        char    rf_data[RF_DATALEN];    /* array containing data to be  */
```

```
                                        /*   written to the file        */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_wres     {               /* remote file write response   */
        RF_MSG_HDR                      /* header fields                */
        uint32  rf_pos;                 /* original position in file    */
        uint32  rf_len;                 /* number of bytes written      */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Open                                    */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_oreq     {               /* remote file open request     */
        RF_MSG_HDR                      /* header fields                */
        int32   rf_mode;                /* Xinu mode bits               */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_ores     {               /* remote file open response    */
        RF_MSG_HDR                      /* header fields                */
        int32   rf_mode;                /* Xinu mode bits               */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Size                                    */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_sreq     {               /* remote file size request     */
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
```

```
struct  rf_msg_sres     {               /* remote file status response  */
        RF_MSG_HDR                      /* header fields                */
        uint32  rf_size;                /* size of file in bytes        */
};
#pragma pack()

/*************************************************************************/
/*                                                                       */
/*                              Delete                                   */
/*                                                                       */
/*************************************************************************/

#pragma pack(2)
struct  rf_msg_dreq     {               /* remote file delete request   */
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_dres     {               /* remote file delete response  */
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()

/*************************************************************************/
/*                                                                       */
/*                              Truncate                                 */
/*                                                                       */
/*************************************************************************/

#pragma pack(2)
struct  rf_msg_treq     {               /* remote file truncate request */
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_tres     {               /* remote file truncate response*/
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()

/*************************************************************************/
/*                                                                       */
/*                              Mkdir                                    */
```

```
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_mreq     {               /* remote file mkdir request    */
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_mres     {               /* remote file mkdir response   */
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Rmdir                                   */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_xreq     {               /* remote file rmdir request    */
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_xres     {               /* remote file rmdir response   */
        RF_MSG_HDR                      /* header fields                */
};
#pragma pack()
```

In the file, constants that begin *RF_MSG_* define a unique type value for each message. For example, *RF_MSG_RREQ* defines the type value used in a *read request* message, and *RF_MSG_RRES* defines the type value used in a *read response* message. The implementation uses a trick to improve efficiency: rather than define arbitrary integers, the type of a response is formed by a logical or of the request type and constant *RF_MSG_RESPONSE*, which is defined to be 0x0100. That is, a response has the same type value as a request except that the low-order bit of the second byte is turned on.

The size of a message depends on the type. Many of the messages only need fields in the common header. For example, a file deletion request only requires a type (to indicate that it is a deletion request), a file name, and a sequence number. Thus, the struct that defines a deletion request, *rf_msg_dreq*, only contains header fields. How-

ever, a *write request* message must include a file offset, the number of data bytes in the request, and the data to be written. Consequently, the struct that defines a *write request* message, *rf_msg_wreq*, includes three additional fields beyond the common header.

## 20.5 Remote File Server Communication

Our remote file system software follows a principle that works well in many cases: the functionality is separated into two levels of software. A lower level handles details of communication with the remote server — it sends a message, waits for a response, and handles retransmission, if necessary. An upper level handles message semantics — it forms a message, passes the message to the lower level for transmission, receives a response, and interprets the response. The important idea is that because it only handles transmission and reception, the lower level does not need to understand or interpret the contents of a message. Consequently, a single function provides all lower-level functionality.

Examining the code will clarify the idea. Function *rfscomm* performs the action of sending a message to the remote file server and receiving a response. File *rfscomm.c* contains the code:

```
/* rfscomm.c - rfscomm */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfscomm - handle communication with RFS server (send request and
 *              receive a reply, including sequencing and retries)
 *------------------------------------------------------------------------
 */
int32   rfscomm (
          struct rf_msg_hdr *msg,        /* message to send              */
          int32  mlen,                   /* message length               */
          struct rf_msg_hdr *reply,      /* buffer for reply             */
          int32  rlen                    /* size of reply buffer         */
        )
{
        int32   i;                       /* counts retries               */
        int32   retval;                  /* return value                 */
        int32   seq;                     /* sequence for this exchange   */
        int16   rtype;                   /* reply type in host byte order*/

        /* For the first time after reboot, register the server port */

        if ( ! Rf_data.rf_registered ) {
```

```
        retval = udp_register(0, Rf_data.rf_ser_port,
                        Rf_data.rf_loc_port);
        Rf_data.rf_registered = TRUE;
}

/* Assign message next sequence number */

seq = Rf_data.rf_seq++;
msg->rf_seq = htonl(seq);

/* Repeat RF_RETRIES times: send message and receive reply */

for (i=0; i<RF_RETRIES; i++) {

        /* Send a copy of the message */

        retval = udp_send(Rf_data.rf_ser_ip, Rf_data.rf_ser_port,
                NetData.ipaddr, Rf_data.rf_loc_port, (char *)msg,
                mlen);
        if (retval == SYSERR) {
                kprintf("Cannot send to remote file server\n\r");
                return SYSERR;
        }

        /* Receive a reply */

        retval = udp_recv(0, Rf_data.rf_ser_port,
                Rf_data.rf_loc_port, (char *)reply, rlen,
                RF_TIMEOUT);

        if (retval == TIMEOUT) {
                continue;
        } else if (retval == SYSERR) {
                kprintf("Error reading remote file reply\n\r");
                return SYSERR;
        }

        /* Verify that sequence in reply matches request */

        if (ntohl(reply->rf_seq) != seq) {
                continue;
        }

        /* Verify the type in the reply matches the request */
```

```
                rtype = ntohs(reply->rf_type);
                if (rtype != ( ntohs(msg->rf_type) | RF_MSG_RESPONSE) ) {
                        continue;
                }

                return retval;          /* return length to caller */
        }

        /* Retries exhausted without success */

        kprintf("Timeout on exchange with remote file server\n\r");
        return TIMEOUT;
}
```

The four arguments specify the address of a message that should be sent to the server, the length of the message, the address of a buffer that will hold a response message, and the length of the buffer. After assigning a unique sequence number to the message, *rfscomm* enters a loop that iterates *RF_RETRIES* times. On each iteration, *rfscomm* uses function *udp_send* to send a copy of the request message over the network† and function *udp_recv* to receive a response.

*Udp_recv* allows a caller to specify a maximum time to wait for a response; *rfscomm* specifies *RF_TIMEOUT*.‡ If no message arrives within the specified time, *udp_recv* returns the value *TIMEOUT*, and the loop continues by transmitting another copy of the request. If no response arrives after *RF_RETRIES* attempts, *rfscomm* returns *TIMEOUT* to its caller.

If a response does arrive, *rfscomm* verifies that the sequence number matches the sequence number in the outgoing request and the message type in the incoming message is the response for the outgoing request. If either test fails, the server formed the message incorrectly or the message was intended for another client on the network. In either case, *rfscomm* continues the loop, sending another copy of the request and waiting for a response to arrive. If the two tests succeed, the incoming message is a valid response, and *rfscomm* returns the length of the response to its caller.

## 20.6 Sending A Basic Message

To understand how *rfscomm* functions, consider a message that only requires the common header fields. For example, the request and response messages used for a *truncate* operation consist of a message header. Because multiple message types only have the common header fields, function *rfsndmsg* has been created to send such a message. File *rfsndmsg.c* contains the code.

†We say that *rfscomm* sends a copy of the message because the original message remains unchanged.

‡*RF_TIMEOUT* is defined to be 1000 milliseconds (i.e., one second), which is ample time for a client to transmit a message across a network to a server and a server to send a response back to the client.

```
/* rfsndmsg.c - rfsndmsg */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfsndmsg - Create and send a message that only has header fields
 *------------------------------------------------------------------------
 */
status  rfsndmsg (
         uint16 type,                   /* message type                 */
         char   *name                   /* null-terminated file name    */
        )
{
        struct  rf_msg_hdr  req;        /* request message to send      */
        struct  rf_msg_hdr resp;        /* buffer for response          */
        int32   retval;                 /* return value                 */
        char    *to;                    /* used during name copy        */

        /* Form a request */

        req.rf_type = htons(type);
        req.rf_status = htons(0);
        req.rf_seq = 0;                 /* rfscomm will set sequence    */
        to = req.rf_name;
        while ( (*to++ = *name++) ) {   /* copy name to request         */
                ;
        }

        /* Send message and receive response */

        retval = rfscomm(&req,  sizeof(struct rf_msg_hdr),
                         &resp, sizeof(struct rf_msg_hdr) );

        /* Check response */

        if (retval == SYSERR) {
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file server access\n\r");
                return SYSERR;
        } else if (ntohl(resp.rf_status) != 0) {
                return SYSERR;
        }

        return OK;
}
```

*Rfsndmsg* takes two arguments that specify the type of the message to send and the name of a file. To create a request message, the code assigns a value to each field of variable *req*. It then calls *rfscomm* to transmit the message and receive a response. If *rfscomm* reports an error or timeout or if the status in the response indicates an error, *rfsndmsg* returns *SYSERR* to its caller. Otherwise, *rfsndmsg* returns *OK*.

## 20.7 Network Byte Order

Remote file access raises an important consideration: the format of integers (i.e., endianness) depends on the computer architecture. If one were to transfer an integer from the memory of one computer directly to the memory on another, the numeric value of the integer on the second computer may differ from the numeric value on the first. To accommodate differences, software that sends data over a computer network follows the convention of converting integers from the local byte order to a standard known as *network byte order*, and software that receives data from a computer network converts integers from network byte order to the local byte order. We can summarize:

> *To accommodate differences in endianness, an integer value sent from one computer to another is converted to network byte order before sending and converted to local byte order upon reception. In our design, upper-level functions perform the conversion.*

Xinu follows the Unix naming convention for byte-order transform functions. Function *htonl* (*htons*) transforms an integer (a short integer) from local host byte order to network byte order; function *ntohl* (*ntohs*) transforms an integer (a short integer) from network byte order to local byte order. For example, function *rfsndmsg* uses *htons* to convert the integers that specify the message type and status from local byte order to network byte order.

## 20.8 A Remote File System Using A Device Paradigm

As we have seen, Xinu uses a device paradigm for both devices and files. The remote file system follows the pattern. Figure 20.1 shows an excerpt from the Xinu *Configuration* file that defines the type of a remote file system master device and a set of remote file pseudo-devices.

```
/* Remote File System master device type */

rfs:
        on udp
                -i rfsInit      -o rfsOpen      -c ioerr
                -r ioerr        -g ioerr        -p ioerr
                -w ioerr        -s ioerr        -n rfsControl
                -intr NULL

/* Remote file pseudo-device type */

rfl:
        on rfs
                -i rflInit      -o ioerr        -c rflClose
                -r rflRead      -g rflGetc      -p rflPutc
                -w rflWrite     -s rflSeek      -n ioerr
                -intr NULL
```

**Figure 20.1** Excerpt from a Xinu Configuration file that defines the two device types used by the remote file system.

Figure 20.2 contains an excerpt from the Configuration file that defines a remote file system master device (*RFILESYS*) and a set of six remote file pseudo-devices (*RFILE0* through *RFILE5*).

```
/* Remote file system master device (one per system) */

                RFILESYS is rfs on udp

/* Remote file pseudo-devices (many instances per system) */

                RFILE0 is rfl on rfs
                RFILE1 is rfl on rfs
                RFILE2 is rfl on rfs
                RFILE3 is rfl on rfs
                RFILE4 is rfl on rfs
                RFILE5 is rfl on rfs
```

**Figure 20.2** Excerpt from a Xinu Configuration file that defines devices used by the remote file system.

When an application calls *open* on the remote file system master device, the call allocates one of the remote file pseudo-devices and returns the device ID of the allocated pseudo-device. The application uses the device ID in calls to *read* and *write*, and eventually calls *close* to deallocate the pseudo-device. The next sections define the device driver functions used for both the remote file system master device and the remote file pseudo-devices.

## 20.9 Opening A Remote File

To open a remote file, a program calls *open* on device *RFILESYS*, supplying a file name and mode argument. *Open* invokes function *rfsOpen*, which forms a request and uses *rfscomm* to communicate with the remote file server. If it succeeds, the call to *open* returns the descriptor of a remote file pseudo-device that is associated with the open file (i.e., can be used to write data into the file or read data from the file). File *rfsOpen.c* contains the code:

```
/* rfsOpen.c - rfsOpen */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfsOpen - allocate a remote file pseudo-device for a specific file
 *------------------------------------------------------------------------
 */

devcall rfsOpen (
         struct dentry  *devptr,        /* entry in device switch table */
         char   *name,                  /* file name to use             */
         char   *mode                   /* mode chars: 'r' 'w' 'o' 'n'  */
        )
{
        struct  rflcblk *rfptr;         /* ptr to control block entry   */
        struct  rf_msg_oreq msg;        /* message to be sent           */
        struct  rf_msg_ores resp;       /* buffer to hold response      */
        int32   retval;                 /* return value from rfscomm    */
        int32   len;                    /* counts chars in name         */
        char    *nptr;                  /* pointer into name string     */
        char    *fptr;                  /* pointer into file name       */
        int32   i;                      /* general loop index           */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);
```

```
        /* Search control block array to find a free entry */

        for(i=0; i<Nrfl; i++) {
                rfptr = &rfltab[i];
                if (rfptr->rfstate == RF_FREE) {
                        break;
                }
        }
        if (i >= Nrfl) {                /* No free table slots remain   */
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Copy name into free table slot */

        nptr = name;
        fptr = rfptr->rfname;
        len = 0;
        while ( (*fptr++ = *nptr++) != NULLCH) {
                len++;
                if (len >= RF_NAMLEN) { /* File name is too long        */
                        signal(Rf_data.rf_mutex);
                        return SYSERR;
                }
        }

        /* Verify that name is non-null */

        if (len==0) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Parse mode string */

        if ( (rfptr->rfmode = rfsgetmode(mode)) == SYSERR ) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Form an open request to create a new file or open an old one */

        msg.rf_type = htons(RF_MSG_OREQ);/* Request a file open         */
        msg.rf_status = htons(0);
        msg.rf_seq = 0;                 /* rfscomm fills in seq. number */
```

```
        nptr = msg.rf_name;
        memset(nptr, NULLCH, RF_NAMLEN);/* initialize name to zero bytes*/
        while ( (*nptr++ = *name++) != NULLCH ) { /* copy name to req.  */
                ;
        }
        msg.rf_mode = htonl(rfptr->rfmode); /* Set mode in request      */

        /* Send message and receive response */

        retval = rfscomm((struct rf_msg_hdr *)&msg,
                                        sizeof(struct rf_msg_oreq),
                        (struct rf_msg_hdr *)&resp,
                                        sizeof(struct rf_msg_ores) );

        /* Check response */

        if (retval == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file open\n\r");
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (ntohs(resp.rf_status) != 0) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Set initial file position */

        rfptr->rfpos = 0;

        /* Mark state as currently used */

        rfptr->rfstate = RF_USED;

        /* Return device descriptor of newly created pseudo-device */

        signal(Rf_data.rf_mutex);
        return rfptr->rfdev;
}
```

Before proceeding to check its arguments, *rfsOpen* checks the remote devices to insure that one is available. The code then checks the file name to insure that the name is

less than the maximum allowed and the mode string to insure that the specification is valid.

Before it allocates the remote file device, *rfsOpen* must communicate with the remote server to insure the file can be opened. The code creates a request message, and uses *rfscomm* to send the message to the server. If a positive response arrives, *rfsOpen* marks the control block entry for the remote file device as being used, sets the initial file position to zero, and returns the descriptor to the caller.

## 20.10 Checking The File Mode

When it needs to check the file mode argument, *rfsOpen* calls function *rfsgetmode*, passing the mode string as an argument. The code can be found in file *rfsgetmode.c*:

```
/* rfsgetmode.c - rfsgetmode */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfsgetmode - parse mode argument and generate integer of mode bits
 *------------------------------------------------------------------------
 */

int32   rfsgetmode (
         char   *mode                   /* string of mode characters    */
        )
{
        int32   mbits;                  /* mode bits to return (in host */
                                        /*    byte order)               */
        char    ch;                     /* next character in mode string*/

        mbits = 0;
        while ( (ch = *mode++) != NULLCH) {
                switch (ch) {

                    case 'r':   if (mbits&RF_MODE_R) {
                                        return SYSERR;
                                }
                                mbits |= RF_MODE_R;
                                continue;

                    case 'w':   if (mbits&RF_MODE_W) {
                                        return SYSERR;
                                }
```

```
                                mbits |= RF_MODE_W;
                                continue;

                        case 'o':   if (mbits&RF_MODE_O || mbits&RF_MODE_N) {
                                        return SYSERR;
                                }
                                mbits |= RF_MODE_O;
                                break;

                        case 'n':   if (mbits&RF_MODE_O || mbits&RF_MODE_N) {
                                        return SYSERR;
                                }
                                mbits |= RF_MODE_N;
                                break;

                        default:    return SYSERR;
                    }
        }

        /* If neither read nor write specified, allow both */

        if ( (mbits&RF_MODE_RW) == 0 ) {
                mbits |= RF_MODE_RW;
        }
        return mbits;
}
```

*Rfsgetmode* extracts characters from the mode string, insures each is valid, and checks for illegal combinations (e.g., a mode string cannot specify both *new* and *old* modes). As it parses the mode string, *rfsgetmode* sets the bits in integer *mbits*. Once it has finished examining the string and checking the combinations, *rfsgetmode* returns integer *mbits* to the caller.

## 20.11 Closing A Remote File

Once a process has finished using a file, the process can call *close* to release the remote file device and make it available for the system to use for another file. For a remote file device, *close* invokes *rflClose*. In our implementation, closing a remote file is trivial. Function *rflClose.c* contains the code:

```
/* rflClose.c - rflClose */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rflClose - Close a remote file device
 *------------------------------------------------------------------------
 */
devcall rflClose (
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  rflcblk *rfptr;         /* pointer to control block     */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Verify remote file device is open */

        rfptr = &rfltab[devptr->dvminor];
        if (rfptr->rfstate == RF_FREE) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Mark device closed */

        rfptr->rfstate = RF_FREE;
        signal(Rf_data.rf_mutex);
        return OK;
}
```

After verifying that the device is currently open, *rflClose* sets the state of the control block entry to *RF_FREE*. Note that this version of *rflClose* does not inform the remote file server that the file is closed. The exercises suggest redesigning the system to inform the remote server when a file is closed.

## 20.12 Reading From A Remote File

Once a remote file has been opened, a process can read data from the file. Driver function *rflRead* performs the *read* operation. The code can be found in file *rflRead.c*:

```
/* rflRead.c - rflRead */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rflRead - Read data from a remote file
 *------------------------------------------------------------------------
 */
devcall rflRead (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer of bytes              */
          int32 count                   /* count of bytes to read       */
        )
{
        struct  rflcblk *rfptr;         /* pointer to control block     */
        int32   retval;                 /* return value                 */
        struct  rf_msg_rreq  msg;       /* request message to send      */
        struct  rf_msg_rres resp;       /* buffer for response          */
        int32   i;                      /* counts bytes copied          */
        char    *from, *to;             /* used during name copy        */
        int32   len;                    /* length of name               */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Verify count is legitimate */

        if ( (count <= 0) || (count > RF_DATALEN) ) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Verify pseudo-device is in use */

        rfptr = &rfltab[devptr->dvminor];

        /* If device not currently in use, report an error */

        if (rfptr->rfstate == RF_FREE) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Verify pseudo-device allows reading */
```

```
if ((rfptr->rfmode & RF_MODE_R) == 0) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Form read request */

msg.rf_type = htons(RF_MSG_RREQ);
msg.rf_status = htons(0);
msg.rf_seq = 0;                 /* rfscomm will set sequence    */
from = rfptr->rfname;
to = msg.rf_name;
memset(to, NULLCH, RF_NAMLEN);  /* start name as all zero bytes */
len = 0;
while ( (*to++ = *from++) ) {   /* copy name to request         */
        if (++len >= RF_NAMLEN) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
}
msg.rf_pos = htonl(rfptr->rfpos);/* set file position           */
msg.rf_len = htonl(count);      /* set count of bytes to read   */

/* Send message and receive response */

retval = rfscomm((struct rf_msg_hdr *)&msg,
                                sizeof(struct rf_msg_rreq),
                 (struct rf_msg_hdr *)&resp,
                                sizeof(struct rf_msg_rres) );

/* Check response */

if (retval == SYSERR) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
} else if (retval == TIMEOUT) {
        kprintf("Timeout during remote file read\n\r");
        signal(Rf_data.rf_mutex);
        return SYSERR;
} else if (ntohs(resp.rf_status) != 0) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}
```

```
        /* Copy data to application buffer and update file position */

        for (i=0; i<htonl(resp.rf_len); i++) {
                *buff++ = resp.rf_data[i];
        }
        rfptr->rfpos += htonl(resp.rf_len);

        signal(Rf_data.rf_mutex);
        return htonl(resp.rf_len);
}
```

*RflRead* begins by checking argument *count* to verify that the request is in range. It then verifies that the pseudo-device has been opened and the mode allows reading. Once the checking is complete, *rflRead* performs the *read* operation: it forms a message, uses *rfscomm* to transmit a copy to the server and receive a response, and interprets the response.

If *rfscomm* returns a valid response, the message will include the data that has been read. *RflRead* copies the data from the response message into the caller's buffer, updates the file position, and returns the number of bytes to the caller.

## 20.13 Writing To A Remote File

Writing to a remote file follows the same general paradigm as reading from a remote file. Driver function *rflWrite* performs the *write* operation; the code can be found in file *rflWrite.c*:

```
/* rflWrite.c - rflWrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rflWrite - Write data to a remote file
 *------------------------------------------------------------------------
 */
devcall rflWrite (
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer of bytes              */
          int32 count                   /* count of bytes to write      */
        )
{
        struct  rflcblk *rfptr;         /* pointer to control block     */
        int32   retval;                 /* return value                 */
        struct  rf_msg_wreq  msg;       /* request message to send      */
```

```
struct  rf_msg_wres resp;       /* buffer for response          */
char    *from, *to;             /* used to copy name            */
int     i;                      /* counts bytes copied into req */
int32   len;                    /* length of name               */

/* Wait for exclusive access */

wait(Rf_data.rf_mutex);

/* Verify count is legitimate */

if ( (count <= 0) || (count > RF_DATALEN) ) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Verify pseudo-device is in use and mode allows writing */

rfptr = &rfltab[devptr->dvminor];
if ( (rfptr->rfstate == RF_FREE) ||
     ! (rfptr->rfmode & RF_MODE_W) ) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Form write request */

msg.rf_type = htons(RF_MSG_WREQ);
msg.rf_status = htons(0);
msg.rf_seq = 0;                 /* rfscomm will set sequence    */
from = rfptr->rfname;
to = msg.rf_name;
memset(to, NULLCH, RF_NAMLEN);  /* start name as all zero bytes */
len = 0;
while ( (*to++ = *from++) ) {   /* copy name to request         */
        if (++len >= RF_NAMLEN) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
}
while ( (*to++ = *from++) ) {   /* copy name into request       */
        ;
}
msg.rf_pos = htonl(rfptr->rfpos);/* set file position           */
msg.rf_len = htonl(count);      /* set count of bytes to write  */
```

```
        for (i=0; i<count; i++) {       /* copy data into message       */
                msg.rf_data[i] = *buff++;
        }
        while (i < RF_DATALEN) {
                msg.rf_data[i++] = NULLCH;
        }

        /* Send message and receive response */

        retval = rfscomm((struct rf_msg_hdr *)&msg,
                                        sizeof(struct rf_msg_wreq),
                        (struct rf_msg_hdr *)&resp,
                                sizeof(struct rf_msg_wres) );

        /* Check response */

        if (retval == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file read\n\r");
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (ntohs(resp.rf_status) != 0) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Report results to caller */

        rfptr->rfpos += ntohl(resp.rf_len);

        signal(Rf_data.rf_mutex);
        return ntohl(resp.rf_len);
}
```

As with a *read* operation, *rflWrite* begins by checking the *count* argument, verifying that the pseudo-device is open and the mode allows writing. *RflWrite* then forms a request message and uses *rfscomm* to send the message to the server.

Unlike a *read* request, a *write* request contains data. Thus, when forming the request, *rflWrite* copies data from the user's buffer into the request message. When a response arrives, the response message does not contain a copy of the data that has been written. Thus, *rflWrite* uses the status field in the message to determine whether to report success or failure to the caller.

## 20.14 Seek On A Remote File

How should a *seek* operation be implemented for our remote file system? There are two possibilities. In one design, the system sends a message to the remote server and the remote server seeks to the specified location in the file. In the other design, all location data is kept on the local computer and each request to the server contains an explicit file position.

Our implementation uses the latter: the current file position is stored in the control block entry for a remote file device. When *read* is called, *rflRead* requests data from the server and updates the file position in the control block entry accordingly. The remote server does not record a position because each request includes explicit position information.

Because all file position information is stored on the client, a *seek* operation can be performed locally. That is, the software stores the file position in the control block entry for use on the next *read* or *write* operation. Function *rflSeek* performs the *seek* operation on a remote file device. The code can be found in file *rflSeek.c*:

```
/* rflSeek.c - rflSeek */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rflSeek - change the current position in an open file
 *------------------------------------------------------------------------
 */
devcall rflSeek (
          struct dentry *devptr,        /* entry in device switch table */
          uint32 pos                    /* new file position            */
        )
{
        struct  rflcblk *rfptr;         /* pointer to control block     */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Verify remote file device is open */

        rfptr = &rfltab[devptr->dvminor];
        if (rfptr->rfstate == RF_FREE) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
```

```
        /* Set the new position */

        rfptr->rfpos = pos;
        signal(Rf_data.rf_mutex);
        return OK;
}
```

The code is trivial. After obtaining exclusive access, *rflSeek* verifies that the device has been opened. It then stores the file position argument in field *rfpos* of the control block, signals the mutual exclusion semaphore, and returns.

## 20.15 Character I/O On A Remote File

Using a remote file server to read and write individual bytes of data is expensive because a message must be sent to the server for each character. Rather than prohibit character I/O, our implementations of *getc* and *putc* merely invoke the remote file functions *rflRead* and *rflWrite*, respectively. Thus, we allow a programmer to decide whether the cost is reasonable. Files *rflGet.c* and *rflPutc.c* contain the code.

```
/* rflGetc.c - rflGetc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  rflGetc - read one character from a remote file (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall rflGetc(
        struct  dentry  *devptr         /* entry in device switch table */
        )
{
        char    ch;                     /* character to read            */
        int32   retval;                 /* return value                 */

        retval = rflRead(devptr, &ch, 1);

        if (retval != 1) {
                return SYSERR;
        }

        return (devcall)ch;
}
```

```
/* rflPutc.c - rflPutc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  rflPutc - write one character to a remote file (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall rflPutc(
        struct  dentry  *devptr,        /* entry in device switch table */
        char    ch                      /* character to write           */
        )
{
        struct  rflcblk *rfptr;         /* pointer to rfl control block */

        rfptr = &rfltab[devptr->dvminor];

        if (rflWrite(devptr, &ch, 1) != 1) {
                return SYSERR;
        }

        return OK;
}
```

## 20.16 Remote File System Control Functions

Several file operations are needed beyond *open*, *read*, *write*, and *close*. For example, it may be necessary to delete a file. The Xinu remote file system uses the *control* function to implement such functions. The table in Figure 20.3 lists the set of symbolic constants used for control functions along with the meaning of each.

| Constant | Meaning |
|---|---|
| RFS_CTL_DEL | Delete the named file |
| RFS_CTL_TRUNC | Truncate a named file to zero bytes |
| RFS_CTL_MKDIR | Make a directory |
| RFS_CTL_RMDIR | Remove a directory |
| RFS_CTL_SIZE | Return the current size of a file in bytes |

**Figure 20.3** Control functions used with the remote file system.

A *control* operation is performed on device *RFILESYS*, the master device for the remote file system, rather than on an individual remote file device. Driver function *rfsControl* implements the *control* operation; the code can be found in file *rfsControl.c*:

```
/* rfsControl.c - rfsControl */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfsControl - Provide control functions for the remote file system
 *------------------------------------------------------------------------
 */
devcall rfsControl (
         struct dentry  *devptr,        /* entry in device switch table */
         int32  func,                   /* a control function           */
         int32  arg1,                   /* argument #1                  */
         int32  arg2                    /* argument #2                  */
        )
{
        int32   len;                    /* length of name               */
        struct  rf_msg_sreq msg;        /* buffer for size request      */
        struct  rf_msg_sres resp;       /* buffer for size response     */
        struct  rflcblk *rfptr;         /* pointer to entry in rfltab   */
        char    *to, *from;             /* used during name copy        */
        int32   retval;                 /* return value                 */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Check length and copy (needed for size) */

        rfptr = &rfltab[devptr->dvminor];
        from = rfptr->rfname;
        to = msg.rf_name;
        len = 0;
        memset(to, NULLCH, RF_NAMLEN);  /* start name as all zeroes     */
        while ( (*to++ = *from++) ) {   /* copy name to message         */
                len++;
                if (len >= (RF_NAMLEN - 1) ) {
                        signal(Rf_data.rf_mutex);
                        return SYSERR;
                }
        }
```

```
        switch (func) {

        /* Delete a file */

        case RFS_CTL_DEL:
                if (rfsndmsg(RF_MSG_DREQ, (char *)arg1) == SYSERR) {
                        signal(Rf_data.rf_mutex);
                        return SYSERR;
                }
                break;

        /* Truncate a file */

        case RFS_CTL_TRUNC:
                if (rfsndmsg(RF_MSG_TREQ, (char *)arg1) == SYSERR) {
                        signal(Rf_data.rf_mutex);
                        return SYSERR;
                }
                break;



        /* Make a directory */

        case RFS_CTL_MKDIR:
                if (rfsndmsg(RF_MSG_MREQ, (char *)arg1) == SYSERR) {
                        signal(Rf_data.rf_mutex);
                        return SYSERR;
                }
                break;

        /* Remove a directory */

        case RFS_CTL_RMDIR:
                if (rfsndmsg(RF_MSG_XREQ, (char *)arg1) == SYSERR) {
                        signal(Rf_data.rf_mutex);
                        return SYSERR;
                }
                break;

        /* Obtain current file size (non-standard message size) */

        case RFS_CTL_SIZE:

                /* Hand-craft a size request message */
```

```
                msg.rf_type = htons(RF_MSG_SREQ);
                msg.rf_status = htons(0);
                msg.rf_seq = 0;         /* rfscomm will set the seq num */

                /* Send the request to server and obtain a response     */

                retval = rfscomm( (struct rf_msg_hdr *)&msg,
                                        sizeof(struct rf_msg_sreq),
                                  (struct rf_msg_hdr *)&resp,
                                        sizeof(struct rf_msg_sres) );
                if ( (retval == SYSERR) || (retval == TIMEOUT) ) {
                        signal(Rf_data.rf_mutex);
                        return SYSERR;
                } else {
                        signal(Rf_data.rf_mutex);
                        return ntohl(resp.rf_size);
                }

        default:
                kprintf("rfsControl: function %d not valid\n\r", func);
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        signal(Rf_data.rf_mutex);
        return OK;
}
```

For all the control functions, argument *arg1* contains a pointer to a null-terminated file name. After it obtains exclusive access and checks the length of the file name, *rfsControl* uses the function argument to choose among several cases that correspond to file deletion, file truncation, directory creation, directory deletion, or a file size request. In each case, *rfsControl* must send a message to the remote server and receive a response.

Except for a file size request, all messages to the server only include the common header fields. Therefore, for all functions except a size request, *rfsControl* uses function *rfsndmsg* to generate and send a request to the remote server. For a size request, *rfsControl* creates a message in variable *msg*, and uses *rfscomm* to send the message and receive a response. To avoid scanning the file name twice, *rfsControl* copies the file name into the name field of variable *msg* as it checks the length of the name. Thus, no extra copy is needed when *rfsControl* creates a size request. If a valid response arrives to a size request, *rfsControl* extracts the file size from the response, converts it to local byte order, and returns the size to the caller. In all other cases, *rfsControl* returns a status of either *OK* or *SYSERR*.

## 20.17 Initializing The Remote File Data Structure

Because the design includes both a remote file system master device and a set of remote file pseudo-devices, two initialization functions are needed. The first, *rfsInit*, initializes the control block associated with the master device. File *rfsInit.c* contains the code:

```
/* rfsInit.c - rfsInit */

#include <xinu.h>

struct  rfdata  Rf_data;

/*------------------------------------------------------------------------
 *  rfsInit - initialize the remote file system master device
 *------------------------------------------------------------------------
 */
devcall rfsInit(
          struct dentry *devptr         /* entry in device switch table */
        )
{

        /* Choose an initial message sequence number */

        Rf_data.rf_seq = 1;

        /* Set the server IP address, server port, and local port */

        if ( dot2ip(RF_SERVER_IP, &Rf_data.rf_ser_ip) == SYSERR ) {
                panic("invalid IP address for remote file server");
        }
        Rf_data.rf_ser_port = RF_SERVER_PORT;
        Rf_data.rf_loc_port = RF_LOC_PORT;

        /* Create a mutual exclusion semaphore */

        if ( (Rf_data.rf_mutex = semcreate(1)) == SYSERR ) {
                panic("Cannot create remote file system semaphore");
        }

        /* Specify that the server port is not yet registered */

        Rf_data.rf_registered = FALSE;
```

```
        return OK;
}
```

Data for the master device is kept in global variable *Rf_data*. *RfsInit* fills in fields of the structure with the remote server's IP address and UDP port number. It also allocates a mutual exclusion semaphore and stores the semaphore ID in the structure. *RfsInit* sets field *rf_registered* to *FALSE*, indicating that before communication with the server is possible, the UDP port of the server must be registered with the network code.

Function *rflInit* handles initialization of individual remote file devices. The code can be found in file *rflInit.c*:

```
/* rflInit.c - rflInit */

#include <xinu.h>

struct  rflcblk rfltab[Nrfl];           /* rfl device control blocks    */

/*------------------------------------------------------------------------
 *  rflInit - initialize a remote file device
 *------------------------------------------------------------------------
 */
devcall rflInit(
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  rflcblk *rflptr;        /* ptr. to control block entry  */
        int32   i;                      /* walks through name arrary    */

        rflptr = &rfltab[ devptr->dvminor ];

        /* Initialize entry to unused */

        rflptr->rfstate = RF_FREE;
        rflptr->rfdev = devptr->dvnum;
        for (i=0; i<RF_NAMLEN; i++) {
                rflptr->rfname[i] = NULLCH;
        }
        rflptr->rfpos = rflptr->rfmode = 0;
        return OK;
}
```

*RflInit* sets the state of the entry to *RF_FREE* to indicate that the entry is currently unused. It also zeroes the name and mode fields. If the state is marked *RF_FREE*, no

references should occur to other fields of the entry. Placing zeroes in the fields aids debugging.

## 20.18 Perspective

As with a local file system, the most complex decision involved in the design of a remote file system arises from the need to choose a balance between efficiency and sharing. To understand the choice, imagine multiple applications running on multiple computers all sharing a single file. At one extreme, to guarantee last-write semantics on a shared file, each file operation must be sent to the remote server so the requests can be serialized and operations can be applied to the file in the order they occur. At the other extreme, efficiency is maximized when a computer can cache files (or parts of files) and access items from the local cache. The goal is to devise a remote file system that maximizes performance when no sharing occurs, guarantees correctness in the presence of sharing, and transitions gracefully and automatically between the two extremes.

## 20.19 Summary

A remote file access mechanism allows applications running on a client computer to access files stored on a remote server. The example design uses a device paradigm in which an application calls *open* on the remote file system master device to obtain the ID of an individual remote file pseudo-device. The application can then use *read* and *write* on the pseudo-device.

When an application accesses a remote file, the remote file software creates a message, sends the message to the remote file server, waits for a response, and interprets the response. The software transmits each request multiple times in case the network drops a packet or the server is too busy to answer.

Operations such as file deletion, file truncation, creating and removing directories, and determining the current size of a file are handled with the *control* function. As with data transfer operations, each call to *control* results in the transmission of a request message and a response from the server.

## EXERCISES

**20.1** Modify the remote file server and *rflClose*. Arrange to have *rflClose* send a message to the server when a file is closed, and have the server send a response.

**20.2** The underlying protocol limits a *read* request to *RF_DATALEN* bytes, and *rflRead* rejects any call that specifies more. Modify *rflRead* to allow a user to request an arbitrary size, but still limit the size in a request message to *RF_DATALEN* (i.e., don't reject large requests, but limit the data returned to *RF_DATALEN* bytes).

**20.3** As an alternative to the exercise above, devise a system in which *rflRead* permits a caller to specify an arbitrary size *read* and sends multiple requests to the server to satisfy the request.

**20.4** The code in *rflGetc* calls *rflRead* directly. What potential problem does such a direct call introduce? Modify the code to use the device switch table when making the call.

**20.5** Consider an alternative design for the remote file system that improves efficiency. Arrange *rflRead* so it always requests *RF_DATALEN* bytes, even if the caller requests fewer. Place extra bytes in a cache, making them available to subsequent calls.

**20.6** In the previous exercise, what is the chief disadvantage of caching data for subsequent *reads*? (Hint: consider shared access to the server.)

**20.7** Consider what happens if two clients attempt to use the remote file server at the same time. When the clients boot, they each start their packet sequence number at 1, which makes the probability of conflict high. Revise the system to use a random starting sequence number (and revise the server to accept arbitrary sequence numbers).

# NOTES

# Chapter Contents

# 21

# A Syntactic Namespace

*A rose by any other name ...*

— William Shakespeare

## 21.1 Introduction

Chapter 14 outlines a set of device-independent I/O operations, including *read* and *write*, and shows how a device switch table provides an efficient mapping between high-level operations and the driver functions for each device. Later chapters describe how device drivers are organized, and provide examples. The previous chapters illustrate how a file system fits into the device paradigm, and illustrates the concept of pseudo-devices.

This chapter considers a generalization of device names. It explains how names can be viewed syntactically, and shows that both devices and files can be represented in a single unified namespace.

## 21.2 Transparency And A Namespace Abstraction

Transparency forms one of the fundamental principles in operating system design:

> *Whenever possible, applications should remain unaware of implementation details such as the location of an object or its representation.*

For example, when an application creates a new process, the application does not need to know the location of the stack that is allocated. Similarly, when an application opens a local file, the application does not need to know the disk blocks the file occupies.

In terms of file access, the Xinu paradigm seems to violate the principle of transparency because it requires the user to specify a file system name when opening a file. For example, the master device for the local file system is named *LFILESYS*. When a Xinu system includes a remote file system, the violation of transparency becomes obvious: a programmer must also know the master device for the remote file system, *RFILESYS*, and must choose between local and remote files. Furthermore, a file name must be chosen to match the naming convention used on the specified file system.

How can we add transparency to file and device naming? The answer lies in a high-level abstraction called a *namespace*. Conceptually, a namespace provides a uniform set of names that knits together apparently diverse file naming schemes into a single unified whole, allowing users to open files and devices without knowing their location. The Unix operating system uses the file system to provide a namespace abstraction: local files, remote files, and devices are each assigned names in a hierarchical file namespace. For example, the name */dev/console* usually corresponds to the system console device and the name */dev/usb* corresponds to the USB device.

Xinu takes a novel approach to the namespace abstraction by separating the namespace mechanism from the underlying file systems. Furthermore, Xinu uses a *syntactic* approach, which means that the namespace examines names without understanding their meaning. What makes our implementation of the namespace especially fascinating is its combination of simplicity and power. By thinking of names as strings, we can understand their similarity. By using the relationship between prefix strings and trees, we can manipulate names easily. By following the principle of access transparency, we can improve the system dramatically. Adding just a small layer of software to existing mechanisms will allow us to unify naming completely.

Before looking at namespace mechanisms, we will review a few examples of file naming to understand the problem at hand. Following the discussion of file names, we will look at a general purpose syntactic naming scheme, and then examine a simpler, less general solution. Finally, we will examine an implementation of the simplified scheme.

## 21.3 Myriad Naming Schemes

The problem designers face when inventing a namespace is simple: they must glue together myriad unrelated naming schemes, each of which has evolved into a self-contained system. On some systems, file names specify the storage device on which the file resides. On others, the filename includes a suffix that tells the type of the file (older systems used suffixes to specify a version for the file). Other systems map all files into a single flat namespace in which each name is merely a string of alphanumeric characters. The following sections give examples of file names on several systems, and

help the reader understand the types and formats of names our namespace must accommodate.

### 21.3.1 MS-DOS

Names in MS-DOS consist of two parts: a device specification and a file name. Syntactically, an MS-DOS name has the form *X*:*file*, where *X* is a single letter that designates the disk device on which the file resides and *file* is the name of the file. Typically, the letter *C* denotes the system hard disk, which means a name like *C:abc* refers to file *abc* on the hard disk.

### 21.3.2 UNIX

UNIX organizes files into a hierarchical, tree structured directory system. A file name is either relative to the current directory or a full *path name* that specifies a path from the root of the directory tree to a file.

Syntactically, full path names consist of slash-separated components, where each intermediate component specifies a directory and the final component specifies a file. Thus, the UNIX name /*homes*/*xinu*/*x* refers to file *x* in subdirectory *xinu*, which is found in subdirectory *homes*, which is contained in the root directory. The root directory itself is named by a single slash (/). Notice that the prefix /*homes*/*xinu*/ refers to a directory, and that the names of all files in that directory share the prefix.

The importance of the *prefix property* will become apparent later. For now, it is sufficient to remember that the tree structure relates to a name prefix:

> *When components in a file name specify a path through a tree-structured directory, the names of all files that reside in the same directory share a common prefix that denotes the directory.*

### 21.3.3 V System

A research operating system known as the *V system* allowed a user to specify a *context* and a name; the system used the context to resolve the name. The syntax used brackets to enclose the context. Thus, [*ctx*] *abc* refers to file *abc* in context *ctx*. Usually, one thinks of each context as a set of files on a particular remote file server.

### 21.3.4 IBIS

The research operating system, *IBIS*, provides yet another syntax for multiple-machine connections. In IBIS, names have the form *machine*:*path*, where *machine* denotes a particular computer system, and *path* is the file name on that machine (e.g., a UNIX full path name).

## 21.4 Naming System Design Alternatives

We seek a single naming system that provides a unified view of all possible file names, independent of the location of the file or the operating system under which it resides. It seems that a designer could choose between two basic approaches in solving the problem: define yet another file naming scheme, or adopt an existing naming scheme. Surprisingly, the Xinu namespace uses neither of these two approaches. Instead, it adds a syntactic naming mechanism that accommodates many underlying naming schemes, while allowing the user to choose a uniform interface to the naming software. The namespace software maps names that the user supplies into names appropriate for the underlying system.

A naming mechanism that accommodates many underlying schemes has several advantages. First, it allows the designer to integrate existing file systems and devices into a single, uniform namespace, even when implemented by remote servers on a set of heterogeneous systems. Second, it permits designers to add new devices or file systems without requiring recompilation of the application programs that use them. Third, it avoids two unattractive extremes. At one extreme, choosing the simplest naming scheme ensures that all file systems can handle the names, but means that the user cannot take advantage of the complexity offered by some servers. At the other extreme, choosing a naming scheme that encompasses the most complex cases means that an application which takes advantage of the complexity may not be able to run on a less sophisticated file system.

## 21.5 A Syntactic Namespace

To understand how to handle names, think of them syntactically: a name is merely a string of characters. A namespace can be created that transforms strings. The namespace does not need to provide files nor directories, nor does it need to understand the semantics of each underlying file system. Instead, the namespace maps strings from a uniform representation chosen by the user into strings appropriate for each particular subsystem. For example, the namespace might translate the string *alf* into the string *C:a_long_file_name*.

What makes a syntactic namespace powerful? Syntactic manipulation is both natural and flexible. Thus, it is easy to specify and understand as well as easy to adapt to many underlying naming schemes. A user can imagine a consistent set of names and use the namespace software to translate them into the forms required by underlying file systems. For example, suppose a system has access to a local file system that uses MS-DOS naming and a remote file system that uses UNIX full path names. The user might adopt the UNIX full path name syntax for all names, making the local disk names start with */local*. In such a scheme, the name */local/abc* would refer to file *abc* on the local hard drive, while the name */etc/passwd* would refer to a remote file. The namespace must translate */local/abc* into *C:abc* so the local MS-DOS file system can

understand it, but would pass */etc/passwd* on to the remote Unix file system without change.

## 21.6 Patterns And Replacements

Exactly how should a syntactic namespace operate? One convenient method uses a *pattern* string to specify the name syntax and a *replacement* string to specify the mapping. For example, consider the pattern replacement pair:

```
"/local"  "C:"
```

which means "translate all occurrences of the string */local* into the string *C:*".

How should patterns be formed? Patterns that consist of literal strings cannot specify replacement unambiguously. In the previous example, the pattern works well on strings like */local/x*, but it fails on strings like */homes/local/bin* because */local* is an internal substring that should not be changed. To be effective, more powerful patterns are needed. For example, Unix pattern matching tools introduce meta-characters that specify how matching should be performed. A *carat* (sometimes called *up-arrow*) matches the beginning of a string. Thus, the Unix pattern:

```
"^/local"  "C:"
```

specifies that */local* only matches at the beginning of a string. Unfortunately, implementations that allow arbitrary patterns and replacements tend to be cumbersome and the patterns become difficult to read. A more efficient solution is needed.

## 21.7 Prefix Patterns

The problem at hand is to find a useful pattern-replacement mechanism that allows the user to specify how names map onto a subsystem without introducing more complexity than is needed. Before thinking about complex patterns, consider what can be done with patterns that consist of literal strings. The key is to imagine files organized into a hierarchy, and to use the prefix property to understand why patterns should specify prefixes.

In a hierarchy, name prefixes group files into subdirectories, making it easy to define the relationship between names and the underlying file systems or devices. Furthermore, each prefix can be represented by a literal string. The point is:

> *Restricting name replacement to prefixes means it is possible to use literal strings to separate underlying file systems into distinct parts of a name hierarchy.*

## 21.8 Implementation Of A Namespace

A concrete example will clarify the details of how a syntactic namespace uses the pattern-replacement paradigm, and will show how the namespace hides subsystem details. In the example, patterns will consist of fixed strings, and only prefixes will be matched. Later sections discuss alternative implementations and generalizations.

The example implementation of a namespace consists of a pseudo-device, *NAMESPACE*, that programs use to open a named object. An application program invokes *open* on the *NAMESPACE* device, passing a name and mode as arguments. The *NAMESPACE* pseudo-device uses a set of prefix patterns to transform the name into a new name, and then passes the new name to the appropriate underlying device through a call to *open*. We will see that all files and devices can be part of the namespace, meaning that an application never needs to open a device other than the *NAMESPACE* pseudo-device.

The next sections present the namespace software, beginning with declarations of the basic data structures, and culminating in the definition of the *NAMESPACE* pseudo-device. Following the declarations, two functions are presented that transform names according to the prefix patterns. The functions form the basis of the most important piece of namespace software: the function that implements *open* for the *NAMESPACE* pseudo-device.

## 21.9 Namespace Data Structures And Constants

File *name.h* contains declarations for the data structures and constants used in the Xinu namespace.

```
/* name.h */

/* Constants that define the namespace mapping table sizes */

#define NM_PRELEN       64              /* max size of a prefix string  */
#define NM_REPLLEN      96              /* maximum size of a replacement*/
#define NM_MAXLEN       256             /* maximum size of a file name  */
#define NNAMES          40              /* number of prefix definitions */

/* Definition of the name prefix table that defines all name mappings */

struct  nmentry {                       /* definition of prefix table   */
        char    nprefix[NM_PRELEN];     /* null-terminated prefix       */
        char    nreplace[NM_REPLLEN];   /* null-terminated replacement  */
        did32   ndevice;                /* device descriptor for prefix */
};
```

```
extern  struct  nmentry nametab[];      /* table of name mappings       */
extern  int32   nnames;                 /* num. of entries allocated    */
```

The principle data structure is array *nametab*, which holds up to *NNAMES* entries. Each entry consists of a prefix pattern string, a replacement string, and a device ID. External integer *nnames* holds a count of the valid entries in *nametab*.

## 21.10 Adding Mappings To The Namespace Prefix Table

Function *mount* is used to add mappings to the prefix table. As expected, *mount* takes three arguments: a prefix string, a replacement string, and a device ID. File *mount.c* contains the code.

```
/* mount.c - mount, namlen */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  mount  -  add a prefix mapping to the name space
 *------------------------------------------------------------------------
 */
syscall mount(
          char          *prefix,        /* prefix to add                */
          char          *replace,       /* replacement string           */
          did32         device          /* device ID to use             */
)
{
        intmask mask;                   /* saved interrupt mask         */
        struct  nmentry *namptr;        /* pointer to unused table entry*/
        int32   psiz, rsiz;             /* sizes of prefix & replacement*/
        int32   i;                      /* counter for copy loop        */

        mask = disable();

        psiz = namlen(prefix, NM_PRELEN);
        rsiz = namlen(replace, NM_REPLLEN);
        if ((psiz == SYSERR) || (rsiz == SYSERR) || isbaddev(device)) {
                restore(mask);
                return SYSERR;
        }
```

```
        if (nnames >= NNAMES) {         /* if table full return error */
                restore(mask);
                return SYSERR;
        }

        /* Allocate a slot in the table */

        namptr = &nametab[nnames];      /* next unused entry in table   */

        /* Copy prefix and replacement strings and record device ID */

        for (i=0; i<psiz; i++) {        /* copy prefix into table entry */
                namptr->nprefix[i] = *prefix++;
        }

        for (i=0; i<rsiz; i++) {        /* copy replacement into entry  */
                namptr->nreplace[i] = *replace++;
        }

        namptr->ndevice = device;       /* record the device ID         */

        nnames++;                       /* increment number of names    */

        restore(mask);
        return OK;
}


/*------------------------------------------------------------------------
 *  namlen  -  compute the length of a string stopping at maxlen
 *------------------------------------------------------------------------
 */
int32   namlen(
          char          *name,          /* name to use                  */
          int32         maxlen          /* maximum length (including a  */
                                        /*   NULL byte)                 */
)
{
        int32   i;                      /* counter */

        /* Search until a null terminator or length reaches max */

        for (i=0; i < maxlen; i++) {
                if (*name++ == NULLCH) {
                        return i+1;
```

```
                }
        }
        return SYSERR;
}
```

If any of the arguments are invalid or the table is full, *mount* returns *SYSERR*. Otherwise, it increments *nnames* to allocate a new entry in the table and fills in the values.

## 21.11 Mapping Names With The Prefix Table

Once a prefix table has been created, name translation can be performed. Mapping consists of finding a prefix match and substituting the corresponding replacement string. Function *nammap* performs translation. The code can be found in file *nammap.c*:

```
/* nammap.c - nammap, namrepl, namcpy */

#include <xinu.h>

status  namcpy(char *, char *, int32);
did32   namrepl(char *, char[]);

/*------------------------------------------------------------------------
 *  nammap  -  using namespace, map name to new name and new device
 *------------------------------------------------------------------------
 */
devcall nammap(
         char   *name,                  /* a name to map                */
         char   newname[NM_MAXLEN],     /* buffer for mapped name       */
         did32  namdev                  /* ID of the namespace device   */
        )
{
        did32   newdev;                 /* device descriptor to return  */
        char    tmpname[NM_MAXLEN];     /* temporary buffer for name    */
        int32   iter;                   /* number of iterations         */

        /* Place original name in temporary buffer and null terminate */

        if (namcpy(tmpname, name, NM_MAXLEN) == SYSERR) {
                return SYSERR;
        }
```

```
        /* Repeatedly substitute the name prefix until a non-namespace  */
        /* device is reached or an iteration limit is exceeded          */

        for (iter=0; iter<nnames ; iter++) {
                newdev = namrepl(tmpname, newname);
                if (newdev != namdev) {
                        namcpy(tmpname, newname, NM_MAXLEN);
                        return newdev;  /* either valid ID or SYSERR    */
                }
        }
        return SYSERR;
}

/*------------------------------------------------------------------------
 *  namrepl  -  use the name table to perform prefix substitution
 *------------------------------------------------------------------------
 */
did32   namrepl(
         char   *name,                  /* original name                */
         char   newname[NM_MAXLEN]      /* buffer for mapped name       */
        )
{
        int32   i;                      /* iterate through name table   */
        char    *pptr;                  /* walks through a prefix       */
        char    *rptr;                  /* walks through a replacement  */
        char    *optr;                  /* walks through original name  */
        char    *nptr;                  /* walks through new name       */
        char    olen;                   /* length of original name      */
                                        /*  including the NULL byte     */
        int32   plen;                   /* length of a prefix string    */
                                        /*  *not* including NULL byte   */
        int32   rlen;                   /* length of replacment string  */
        int32   remain;                 /* bytes in name beyond prefix  */
        struct  nmentry *namptr;        /* pointer to a table entry     */

        /* Search name table for first prefix that matches */

        for (i=0; i<nnames; i++) {
                namptr = &nametab[i];
                optr = name;            /* start at beginning of name   */
                pptr = namptr->nprefix; /* start at beginning of prefix */
```

```
                /* Compare prefix to string and count prefix size */

                for (plen=0; *pptr != NULLCH ; plen++) {
                        if (*pptr != *optr) {
                                break;
                        }
                        pptr++;
                        optr++;
                }
                if (*pptr != NULLCH) {  /* prefix does not match */
                        continue;
                }

                /* Found a match - check that replacement string plus   */
                /* bytes remaining at the end of the original name will */
                /* fit into new name buffer.  Ignore null on replacement*/
                /* string, but keep null on remainder of name.          */

                olen = namlen(name ,NM_MAXLEN);
                rlen = namlen(namptr->nreplace,NM_MAXLEN) - 1;
                remain = olen - plen;
                if ( (rlen + remain) > NM_MAXLEN) {
                        return (did32)SYSERR;
                }

                /* Place replacement string followed by remainder of */
                /* original name (and null) into the new name buffer */


                nptr = newname;
                rptr = namptr->nreplace;
                for (; rlen>0 ; rlen--) {
                        *nptr++ = *rptr++;
                }
                for (; remain>0 ; remain--) {
                        *nptr++ = *optr++;
                }
                return namptr->ndevice;
        }
        return (did32)SYSERR;
}
```

```
/*------------------------------------------------------------------------
 *  namcpy  -  copy a name from one buffer to another, checking length
 *------------------------------------------------------------------------
 */
status  namcpy(
         char           *newname,       /* buffer to hold copy          */
         char           *oldname,       /* buffer containing name       */
         int32          buflen          /* size of buffer for copy      */
        )
{
        char    *nptr;                  /* point to new name            */
        char    *optr;                  /* point to old name            */
        int32   cnt;                    /* count of characters copied   */

        nptr = newname;
        optr = oldname;

        for (cnt=0; cnt<buflen; cnt++) {
                if ( (*nptr++ = *optr++) == NULLCH) {
                        return OK;
                }
        }
        return SYSERR;          /* buffer filled before copy completed  */
}
```

The most interesting aspect of *nammap* arises because it allows multiple mappings. In particular, because the namespace is a pseudo-device, it is possible for a user to specify a mapping back onto the *NAMESPACE* device. For example, consider the following two entries in *nametab*:

```
"/local/"    ""           LFILESYS
"LFS:"       "/local/"    NAMESPACE
```

The first entry specifies that if a name begins with */local/*, the prefix is removed and the name is passed to the local file system. The second entry specifies that *LFS:* is an abbreviation for */local/*. That is, the prefix *LFS:* is replaced by */local/* and the resulting string is passed back to the NAMESPACE device for another round of mapping.

Of course, recursive mapping can be dangerous. Consider what can happen if a user adds the following to the namespace:

```
"/x"        "/x"         NAMESPACE
```

When presented with a name */xyz*, a naive implementation will find prefix */x*, make the substitution, and call *open* on the NAMESPACE device, causing an infinite recursion. To avoid the problem, our implementation iterates through NAMESPACE replacements

and limits the total iterations. In particular, the code only permits one iteration for each prefix in *nametab* (i.e., each prefix can be substituted at most once). Of course, *nammap* also limits the size of a name: if a replacement would expand the name to more than *NM_MAXLEN* characters, *nammap* stops and returns *SYSERR*.

*Nammap* begins by copying the original name into local array *tmpname*. It then iterates until the name has been mapped to a device other than the NAMESPACE or the iteration limit is reached. During each iteration, *nammap* calls function *namrepl* to look up the current name and form a replacement.

Function *namrepl* implements a basic *replacement policy*. Our example replacement policy is simplistic: *namrepl* searches the table linearly. A search always begins with the first entry in the table, and stops as soon as a prefix in the table matches the string supplied by argument *name*. Once searching has stopped, *nammap* forms a mapped name in argument *newname* by appending the unmatched portion of the original name onto the replacement string. It then returns the device ID from the table entry. A later section explains that the design has consequences for users.

## 21.12 Opening A Named File

Once *nammap* is available, constructing the upper-half *open* routine for the namespace pseudo-device becomes trivial. Recall that the basic goal is to define a namespace pseudo-device, *NAMESPACE*, such that opening the device causes the system to open the appropriate underlying device. Once a name has been mapped and a new device identified, *namopen* merely invokes *open*. The code is contained in file *namopen.c*.

```
/* namopen.c - namopen */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  namopen  -  open a file or device based on the name
 *------------------------------------------------------------------------
 */
devcall namopen(
          struct dentry *devptr,        /* entry in device switch table */
          char  *name,                  /* name to open                 */
          char  *mode                   /* mode argument                */
        )
{
        char    newname[NM_MAXLEN];     /* name with prefix replaced    */
        did32   newdev;                 /* device ID after mapping      */

        /* Use namespace to map name to a new name and new descriptor */

        newdev = nammap(name, newname, devptr->dvnum);

        if (newdev == SYSERR) {
                return SYSERR;
        }

        /* Open underlying device and return status */

        return  open(newdev, newname, mode);
}
```

## 21.13 Namespace Initialization

How should the prefix table be initialized? There are two possible approaches: provide an initialization mechanism and require a user to fill in entries in the namespace table, or provide initial entries for the table. The mechanism answer is easy. Because the namespace has been designed as a pseudo-device, the system carries out initialization at startup when it calls *init*.

Deciding how to initialize a prefix table can be difficult. Therefore, we will examine the initialization function to see how it constructs a prefix table, and defer the discussion of actual prefixes until later sections. File *naminit.c* contains the code for the *naminit* function:

```
/* naminit.c - naminit */

#include <xinu.h>

#ifndef RFILESYS
#define RFILESYS        SYSERR
#endif

#ifndef FILESYS
#define FILESYS         SYSERR
#endif

#ifndef LFILESYS
#define LFILESYS        SYSERR
#endif

struct  nmentry nametab[NNAMES];        /* table of name mappings       */
int32   nnames;                         /* num. of entries allocated    */

/*------------------------------------------------------------------------
 *  naminit  -  initialize the syntactic namespace
 *------------------------------------------------------------------------
 */
status  naminit(void)
{
        did32   i;                      /* index into devtab            */
        struct  dentry *devptr;         /* ptr to device table entry    */
        char    tmpstr[NM_MAXLEN];      /* string to hold a name        */
        status  retval;                 /* return value                 */
        char    *tptr;                  /* ptr into tempstring          */
        char    *nptr;                  /* ptr to device name           */
        char    devprefix[] = "/dev/";  /* prefix to use for devices    */
        int32   len;                    /* length of created name       */
        char    ch;                     /* storage for a character      */

        /* Set prefix table to empty */

        nnames = 0;

        for (i=0; i<NDEVS ; i++) {
                tptr = tmpstr;
                nptr = devprefix;
```

```
        /* Copy prefix into tmpstr*/

        len = 0;
        while ((*tptr++ = *nptr++) != NULLCH) {
                len++;
        }
        tptr--; /* move pointer to position before NULLCH */
        devptr = &devtab[i];
        nptr = devptr->dvname;  /* move to device name */

        /* Map device name to lower case and append */

        while(++len < NM_MAXLEN) {
                ch = *nptr++;
                if ( (ch >= 'A') && (ch <= 'Z')) {
                        ch += 'a' - 'A';
                }
                if ( (*tptr++ = ch) == NULLCH) {
                        break;
                }
        }

        if (len > NM_MAXLEN) {
                kprintf("namespace: device name %s too long\r\n",
                                devptr->dvname);
                continue;
        }

        retval = mount(tmpstr, NULLSTR, devptr->dvnum);
        if (retval == SYSERR) {
                kprintf("namespace: cannot mount device %d\r\n",
                                devptr->dvname);
                continue;
        }
}

/* Add other prefixes (longest prefix first) */

mount("/dev/null",      "",         CONSOLE);
mount("/remote/",       "remote:", RFILESYS);
mount("/local/",        NULLSTR,    LFILESYS);
mount("/dev/",          NULLSTR,    SYSERR);
mount("~/",             NULLSTR,    LFILESYS);
mount("/",              "root:",    RFILESYS);
mount("",               "",         LFILESYS);
```

```
        return OK;
}
```

Ignore the specific prefix and replacement names and look only at how straightforward initialization is. After setting the number of valid entries to zero, *naminit* calls *mount* to add entries to the prefix table, where each entry contains a prefix pattern, replacement string, and device id. The for loop iterates through the device switch table. For each device, it creates a name of the form */dev/xxx*, where *xxx* is the name of the device mapped into lower case. Thus, it creates an entry for */dev/console* that maps to the CONSOLE device. Thus, if a process calls:

```
d = open(NAMESPACE,"/dev/console","rw");
```

the namespace will invoke *open* on the CONSOLE device and return the result.

## 21.14 Ordering Entries In The Prefix Table

The Xinu name replacement policy affects users. To understand how, recall that *namrepl* uses sequential lookup. Therefore, a user must mount names so that sequential lookup produces the expected outcome. In particular, our implementation does not prohibit overlapping prefixes, and does not warn users if overlaps occur. Consequently, if overlapping prefixes occur, a user must insure that the longest prefix appears earlier in the table than shorter prefixes. As an example, consider what happens if the table contains two entries as Figure 21.1 illustrates.

| Prefix | Replacement | Device |
|---|---|---|
| "x" | "" (null string) | LFILESYS |
| "xyz" | "" (null string) | RFILESYS |

**Figure 21.1** Two entries in a prefix table; the order must be swapped or the second entry will never be used.

The first entry maps prefix *x* to a local file system, and the second entry maps prefix *xyz* to a remote file system. Unfortunately, because *namrepl* searches the table sequentially, any file name that starts with *x* will match the first entry and will be mapped to the local file system. The second entry will never be used. If the two are reversed, however, file names starting with *xyz* will map onto the remote file system, and other names starting with *x* will map to the local file system. The point is:

*Because our implementation searches the table of prefixes sequentially and does not detect overlapping prefixes, a user must insert prefixes in reverse order by length, insuring that the system will match the longest prefix first.*

## 21.15 Choosing A Logical Namespace

It is tempting to think of a namespace as merely a mechanism that can be used to abbreviate long names. However, focusing on the mechanism can be misleading. The key to choosing meaningful prefix names lies in thinking of a hierarchy into which files can be placed. Then, the namespace design defines the organization of the hierarchy.

*Rather than thinking of the namespace as a mechanism that abbreviates names, we think of all names being organized into a hierarchy. Entries in the namespace are chosen to implement the desired hierarchy.*

Imagine, for a minute, a system that can access files on a local disk as well as files on a remote server. Do not think about how to abbreviate specific file names; think instead of how to organize the files. Three possible organizations come to mind as Figure 21.2 shows.

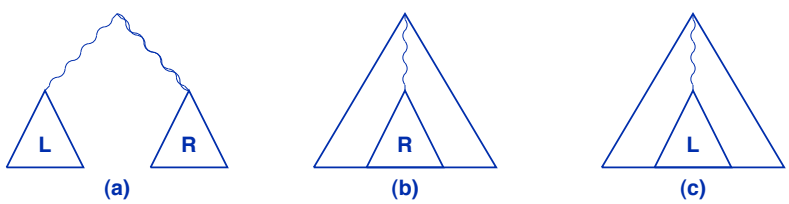



**Figure 21.2** Three possible hierarchical organizations of local and remote files: (a) local and remote files at the same level, (b) remote files in a subdirectory of local files, and (c) local files as a subdirectory of remote files.

As in the figure, local and remote files could be placed at equal, but distinct positions in the hierarchy. Alternatively, the local system could form the main part of the hierarchy with remote files in a sub-hierarchy, or the remote files could form the main hierarchy with local files as a sub-hierarchy. Among the choices, the size of the two file systems and the frequency of access may help determine which organization is preferable. For example, if the remote file system has thousands of files while the local

file system has only ten, it may be natural to think of the remote file system as the main hierarchy with the local files grafted onto a sub-hierarchy.

## 21.16 A Default Hierarchy And The Null Prefix

The Xinu namespace software design can easily support any of the hierarchies shown in Figure 21.2. In particular, *mount* permits the user to choose one subsystem as the default, and organize the remaining files with respect to that hierarchy.

How does a subsystem become the default? First, the prefix for the subsystem must be such that it matches all names not matched by other table entries. The null prefix provides guaranteed matching for our example namespace. Second, the default entry, which carries the null prefix, must be examined only after all other prefixes have been tested. Because *nammap* searches the prefix table sequentially, the default must be placed at the end of the table. If any other entry matches, *namrepl* will follow the match.

Look at *naminit* again to see how the local file system becomes the default. The final call to *mount* inserts the default mapping with a null prefix. Thus, any name that does not match another prefix will refer to the local file system.

## 21.17 Additional Object Manipulation Functions

Although it appears to organize all names into a single, unified hierarchy, the namespace illustrated above does not provide all needed functionality. To understand why, observe that the code only handles the case of opening a named object. Other operations on named objects are also possible:

- Testing the existence of an object
- Changing the name of an object
- Deleting an object

*Testing the existence of an object*. Often software needs to test the existence of an object without otherwise affecting the object. It may seem that the following test could be used to determine whether an object exists.

```
dev = open(NAMESPACE", "object", "r");
if (dev == SYSERR) {
        ...object does not exist
} else {
        close(dev);
        ...object exists
}
```

Unfortunately, a call to *open* can have side effects. For example, opening a network interface device can cause the system to declare the interface to be available for packet transfer. Thus, opening and then closing the device may cause packet transfer to begin, even if a process has specifically disabled the interface. To avoid side effects, additional functionality is needed.

*Changing the name of an object.* Most file systems allow a user to rename files. However, two problems arise when the namespace is used. First, because users view all files through the namespace, requests to rename files can be ambiguous: should the name of the underlying file be changed or should the mapping in the namespace be changed? Although it is possible to include an escape mechanism that allows a user to distinguish between abstract names and names used in the underlying system, doing so is dangerous because changes to underlying names may no longer map through the namespace. Second, if a user specifies changing name $\alpha$ to name $\beta$, it may turn out that string $\beta$ maps to a local file system and $\alpha$ maps to a remote file system. Thus, even though a user sees a unified hierarchy, a renaming operation may not be allowed (or may involve a file copy).

*Deleting an object.* The reasoning given above applies to object deletion as well. That is, because a user views all names through the namespace, a request to delete an object must map the name through the namespace to determine the underlying file system.

How should deletion, renaming, and testing the existence of an object be implemented? Three possibilities arise: separate functions could be created for each operation, the device switch table could be expanded, or the *control* functions could be augmented with additional operations. To understand the first approach, separate functions, imagine a function *DeleteObj* that takes a name as an argument. The function would use the namespace to map the name to an underlying device, and then invoke the appropriate deletion operation on the device. The second approach consists of expanding the device switch table to add additional high-level functions, such as *delete*, *rename*, and an *existence test*. That is, in addition to *open*, *read*, *write*, and *close* operations, add new operations that implement the additional functionality. The third approach consists of adding functionality to the *control* function. For example, we could specify that if a subsystem implements object deletion, the driver function that implements *control* must honor a *DELETE* request. Xinu uses a mixture of the first and third approaches. Exercises ask the reader to consider the advantages and disadvantages of expanding the device switch table.

## 21.18 Advantages And Limits Of The Namespace Approach

A syntactic namespace isolates programs from the underlying devices and file systems, allowing a naming hierarchy to be imagined or changed without changing the underlying systems. To appreciate the power of a namespace, consider a system that keeps temporary files on a local disk and uses the prefix */tmp/* to distinguish them from other files. Moving temporary files to the remote file system consists of changing the

namespace entry that specifies how to handle prefix */tmp/*. Because programs always use the namespace when they refer to files, all programs continue to operate correctly with no change to the source code. The key point is:

> *A namespace permits the conceptual naming hierarchy to be reorganized without requiring recompilation of programs that use it.*

Namespace software that uses only prefix patterns cannot handle all hierarchical organizations or file mappings. For example, in some UNIX systems, the name */dev/tty* refers to a process's control terminal, which a server should not use. The namespace can prevent accidental access by mapping the prefix */dev/tty* onto device ID *SYSERR*. Unfortunately, such a mapping prevents the client from accessing other entries that share the same prefix (e.g., */dev/tty1*).

Using fixed strings as prefix patterns also prevents the namespace from changing separator characters when they occur in the middle of names. For example, suppose a computer has two underlying file systems, one of which follows the UNIX convention of using the slash character to separate components along a path, while the other uses the backslash character to separate components. Because it deals only with prefix patterns, our example namespace cannot map slashes to backslashes or vice versa unless all possible prefixes are stored in the namespace.

## 21.19 Generalized Patterns

Many of the namespace limitations can be overcome by using more general patterns that were described near the beginning of the chapter. For example, if it is possible to specify a *full string match* instead of just a prefix match, the problem of distinguishing a name like */dev/tty* from the name */dev/tty1* can be solved. Full match and prefix match can be combined: *mount* can be modified to have an additional argument that specifies the type of match, and the value can be stored in the table entry.

Generalizing patterns to allow more than fixed strings solves additional problems and keeps all matching information in the pattern itself. For example, suppose characters have special meanings in a pattern as defined in Figure 21.3:†

| Character | Meaning |
|---|---|
| ↑ | match beginning of string |
| $ | match end of string |
| . | match any single character |
| * | repeat 0 or more of a pattern |
| \ | take next character in pattern literally |
| other | self match as in a fixed string |

**Figure 21.3** An example definition of generalized patterns.

†The pattern matching given here corresponds to that used in the UNIX *sed* command.

Thus, a pattern like ↑/*dev*/*tty*$ specifies a full match of the string */dev/tty*, while a pattern like \$ matches a dollar sign that may be embedded in the string.

Two additional rules are necessary to make generalized pattern matching useful in the namespace. First, we assume the left-most possible match will be used. Second, we assume that among all left-most matches, the longest will be selected. The exercises suggest how to use these generalized patterns to map the names that fixed prefixes cannot handle.

## 21.20 Perspective

The syntax of names has been studied and debated extensively. At one time, each operating system had its own naming scheme, and a variety of naming schemes flourished. However, once hierarchical directory systems became prevalent, most operating systems adopted a hierarchical naming scheme, and the only differences arise over small details, such as whether a forward slash or backslash is used to separate components.

As our design points out, naming is conceptually separate from the underlying file and I/O systems, and allows a designer to impose a consistent namespace across all underlying facilities. However, the use of a syntactic approach has disadvantages as well as advantages. The chief problems arise from semantics: although it provides the appearance of uniformity, the namespace introduces ambiguity and confuses semantics. For example, if an object is renamed, should the namespace be modified or should the name of the underlying object be changed? If the namespace maps two prefixes to the same underlying file system, applications that use separate prefixes could inadvertently access the same object. If two names map to different underlying file systems, an operation such as *move* that references the names may not be possible or may not work as expected. Even an operation such as *delete* may have unexpected semantics (e.g., deleting a local object may move it to the trash, while deleting a remote object permanently removes the object).

## 21.21 Summary

Dealing with file names is difficult, especially if the operating system supports multiple underlying naming schemes. One way to solve the naming problem employs a layer of namespace software between applications and the underlying file systems. The namespace does not implement files itself, but merely treats names as strings, mapping them into forms appropriate for underlying systems based on information in a mapping table.

We examined the implementation of a syntactic namespace that uses a pattern-replacement scheme in which patterns are fixed strings representing name prefixes. The software includes functions *mount* and *unmount*† that manipulate the mapping table, as well as functions like *nammap* that map names into their target form. Our example namespace comprises a *NAMESPACE* pseudo-device that users specify when opening a

†Function *unmount*, which is not shown in the chapter, removes a prefix from the namespace.

file. The *NAMESPACE* pseudo-device maps the specified file name, and then opens the designated file.

The namespace software is both elegant and powerful. With only a few functions and the simplistic notion of prefix matching, it can accommodate many naming schemes. In particular, it accommodates a remote file system, a local file system, and a set of devices. However, our simplistic version cannot handle all possible mappings. To provide for more complex naming systems, the notion of pattern must be generalized. One possible generalization assigns special meaning to some characters in the pattern.

## EXERCISES

**21.1** Should users have access to both *nammap* and *namrepl*? Why or why not?

**21.2** Is it possible to modify *mount* so it refuses to mount prefix-replacement pairs that can potentially cause an infinite loop? Why or why not?

**21.3** What is the minimum number of prefix-replacement pairs that can cause *nammap* to exceed the maximum string length?

**21.4** Minimize the code in *namopen* by replacing the body with a single statement consisting of two procedure calls.

**21.5** Implement an upper-half *control* function for the *NAMESPACE* pseudo-device, and make *nammap* a control function.

**21.6** Implement generalized pattern matching. Refer to the UNIX *sed* command for additional ways to define pattern match characters.

**21.7** Build a namespace that has both prefix matches and full string matches.

**21.8** Suppose a namespace uses fixed string patterns, but allows full string matching in addition to the current prefix matching. Are there instances when it makes sense to have a full string pattern identical to a prefix pattern? Explain.

**21.9** What additional file manipulation primitives might be needed beside *rename*, *delete*, and *existence test*?

# Chapter Contents

# 22

# System Initialization

*Only by avoiding the beginning of things can we escape their end.*

— Cyril Connolly

## 22.1 Introduction

Initialization is the last step of the design process. Designers create a system by thinking about the system in an executing state, postponing the details of how to get the system started. Thinking about initialization early has the same bad effect as worrying about optimization early: it tends to impose unnecessary constraints on the design, and diverts the designer's attention from important issues to trivial ones.

This chapter describes the steps required to initialize the system, and explains the conceptual leap that initialization code makes in switching from a program that executes sequentially to a system that supports concurrent processes. We will see that no special hardware steps are involved, and understand that concurrency is an abstraction created entirely by the operating system.

## 22.2 Bootstrap: Starting From Scratch

Our discussion of initialization begins with a consideration of system termination. Everyone who has worked with a computer system knows that errant programs or malfunctions in the hardware lead to catastrophic failures popularly called *crashes*. A crash occurs when the hardware attempts an invalid operation because code or data for a given operation is incorrect. Users know that when a crash occurs, the contents of memory are lost and the operating system must be restarted, which often takes considerable time.

How can a computer that is devoid of operating system code spring into action and begin executing? It cannot. Somehow a program must be available before a computer can start. On the oldest machines, restarting was a painful process because a human operator entered the initial program through switches on the front panel. Switches were replaced by keyboards, then by I/O devices such as tapes and disks, and eventually by Read-Only Memory (ROM).

Some embedded devices do more than store an initial program in ROM; they store the entire operating system, which means that the device can start executing immediately after receiving power (e.g., after the batteries are changed or the device is powered on). However, most computers take multiple steps when restarting. When power is applied, the hardware executes an initial startup program from ROM. Although it may include mechanisms that allow engineers to debug the hardware, an initial program is usually quite small — its primary function consists of loading and running a larger program. In a typical personal computer, for example, the startup program powers on devices (e.g., a display, keyboard, and disk), copies the operating system image from disk into memory, and then jumps to the entry point of the operating system.

Computer systems linked to a network may perform an additional step: the initial startup program may load an intermediate program that initializes the network interface, and then uses the network to download the operating system image from a remote server.

Programs in a sequence that load ever larger programs are often called *bootstraps*, and the entire process is called *booting* the system.† Older names for the bootstrap process include *Initial Program Load* (*IPL*) and *cold start*.

## 22.3 Operating System Initialization

Of course, the work of initialization does not end when the CPU begins to execute the operating system. The system must perform the following tasks:

- Initialize the memory management hardware and the free memory list
- Initialize each operating system module
- Load (if not present) and initialize each of the device drivers
- Start (or reset) each I/O device
- Transform from a sequential program to a concurrent system
- Create a null process
- Create a process to execute user code (e.g., a desktop)

The most important step occurs once basic initialization completes: the operating system must undergo a metamorphosis, changing itself from a program that executes sequentially into an operating system that runs processes and supports concurrent execution. In the next sections, we will see what happens when an E2100L boots, understand

†The terminology derives from the phrase "pulling one's self up by one's bootstraps," a seemingly impossible task.

how Xinu is loaded onto the hardware, review the sequence of steps Xinu takes when execution begins, and see exactly how the transformation occurs.

## 22.4 Booting An Alternative Image On The E2100L

When used as a wireless router, the E2100L follows the steps outlined above. After it is powered on, the router executes an initial startup program in ROM. Once it has initialized devices, the startup program runs a version of embedded Linux (the Linux code is also kept in ROM). After it boots, Linux launches applications that turn the system into a wireless router. For example, one application uses standard network protocols, such as IP, UDP, and DHCP, to talk over an Ethernet connection to an *Internet Service Provider* (*ISP*). Another honors requests that the router receives from local computers over the wired or wireless networks. Once Linux is running, all input and output is controlled by built-in Linux processes, and the device can only act as a router; there is no way to replace or augment the software.

If an E2100L is designed to run a built-in operating system and applications, how can one make it boot Xinu? Fortunately, the vendor has included a way to interrupt the normal startup sequence and gain control before Linux boots. To gain control, one uses the console serial line (which is hidden from the user unless they remove the cover and connect directly to the circuit board). Typing characters on the serial line during bootstrap causes the initial startup program to stop before it loads Linux and to begin accepting commands from the console. That is, the program issues the prompt:

```
ar7100>
```

and expects the user to enter a command. After executing the command, the startup program sends the prompt again and waits for another command.

The startup program honors several commands that are used to download an image. For example, the *loadb* command can be used to download a binary image over the console serial line using the kermit protocol. Fortunately, the E2100L also includes a *bootp* command that uses standard protocols *BOOTP* and *TFTP* to download an image over the network. The *bootp* command takes an argument that specifies where to load the image; Xinu is loaded into kernel space at location *0x81000000* with the command:

```
bootp 0x81000000
```

The E2100L sends a *BOOTP* request packet. The network must have a BOOTP server running that has been configured to answer the request. Among other items, the answer specifies the name of a file to download and the address of a server from which to obtain the file. The E2100L sends a sequence of *TFTP* request packets to the server to obtain the file. The network must also have a TFTP server running that has been configured to respond to each request packet by sending the requested piece of the file.

Once it has downloaded a file and placed the file in memory, the startup program again issues a prompt on the console line and waits for a command. If the user enters the command

```
bootm
```

the startup program branches to location *0x81000000* (i.e., begins running the Xinu code that was downloaded into memory).

## 22.5 Xinu Initialization

Before it can become an operating system, Xinu must establish the runtime environment needed for C, initialize operating system modules, and enable interrupts. On the E2100L, the bootstrap startup program handles some of the low-level hardware initialization tasks. For example, by the time Xinu runs, the startup program has initialized the system bus. The startup program has also initialized the console serial device (e.g., set the baud rate), making it possible for Xinu code to send and receive characters over the console line immediately.

Although some low-level initialization is completed before Xinu boots, an assembly language initialization function is still required. In our code, execution begins at label *_start*, found in file *start.S*:

```
/*  start.S  _start, memzero */

/************************************************************************/
/*                                                                      */
/*   External symbol start (_start in assembly language) gives the      */
/* location where execution begins after the bootstrap loader has       */
/* placed a Xinu image in memory and is ready to execute the image.     */
/*                                                                      */
/*   After initializing the hardware and establishing a run-time        */
/* environment suitable for C (including a valid stack pointer), the    */
/* code jumps to the C function nulluser.                               */
/************************************************************************/

#include <interrupt.h>
#include <mips.h>

#define NULLSTK      8192       /* Safe size for NULLSTK */

.extern flash_size

.text
        .align  4
        .globl  _minheap
        .globl  _start
```

```
/*------------------------------------------------------------------------
 *
 * _start   - set up interrupts, initialize the stack pointer, clear the
 *            null process stack, zero the BSS (uninitialized data)
 *            segment, and invoke nulluser
 *------------------------------------------------------------------------
 */
        .ent _start
_start:

        /* Pick up flash size from a3 (where the boot loader leaves it) */

        sw      a3, flash_size

        /* Clear Xinu-defined trap and interrupt vectors */

        la      a0, IRQ_ADDR
        la      a1, IRQVEC_END
        jal     memzero

        /* Copy low-level interrupt dispatcher to reserved location.    */

        la      a0, IRQ_ADDR            /* Reserved vector location     */
        la      a1, intdispatch         /* Start of dispatch code       */
        lw      v0, 0(a1)
        sw      v0, 0(a0)               /* Store jump opcode            */

        /* Clear interrupt related registers in the coprocessor */

        mtc0    zero, CP0_STATUS        /* Clear interrupt masks        */
        mtc0    zero, CP0_CAUSE         /* Clear interrupt cause reg.   */

        /* Clear and invalidate the L1 instruction and data caches */

        jal     flushcache

        /* Set up Stack segment (see function summary) */

        li      s0, NULLSTK             /* Stack is NULLSTK bytes       */
        la      a0, _end
        addu    s0, s0, a0              /* Top of stack = _end+NULLSTK  */

        /* Word align the top of the stack */
```

```
        subu    s1, s0, 1
        srl     s1, 4
        sll     s1, 4

        /* Initialize the stack and frame pointers */

        move    sp, s1
        move    fp, s1

        /* Zero NULLSTK space below new stack pointer */

        move    a1, s0          /* note; a0 still points to _end */
        jal     memzero

        /* Clear the BSS segment */

        la      a0, _bss
        la      a1, _end
        jal     memzero

        /* Store bottom of the heap */

        la      t0, minheap
        sw      s0, 0(t0)

        j       nulluser        /* jump to the null process code */
        .end    _start

/*------------------------------------------------------------------------
 * memzero - clear a specified area of memory
 *
 *      args are: starting address and ending address
 *------------------------------------------------------------------------
 */
        .ent    memzero
memzero:
        sw      zero, 0(a0)
        addiu   a0, a0, 4
        blt     a0, a1, memzero
        jr      ra
        .end    memzero
```

## 22.6 System Startup

A single program, not an operating system, is running when the processor jumps to the C function *nulluser*. The program initializes important operating system data structures, devices, semaphores, and processes. The code is found in file *initialize.c*. If there is drama in the system, it lies here, where the transformation from program to system begins.

```
/* initialize.c - nulluser, sysinit */

/* Handle system initialization and become the null process */

#include <xinu.h>
#include <string.h>

extern  void    _start(void);   /* start of Xinu code */
extern  void    *_end;          /* end of Xinu code */

/* Function prototypes */

extern  void main(void);        /* main is the first process created    */
extern  void xdone(void);       /* system "shutdown" procedure          */
static  void sysinit(void);     /* initializes system structures        */

/* Declarations of major kernel variables */

struct  procent proctab[NPROC]; /* Process table                        */
struct  sentry  semtab[NSEM];   /* Semaphore table                      */
struct  memblk  memlist;        /* List of free memory blocks           */

/* Active system status */

int     prcount;                /* Total number of live processes       */
pid32   currpid;                /* ID of currently executing process    */

/* Memory bounds set by startup.S */

void    *minheap;               /* start of heap                        */
void    *maxheap;               /* highest valid memory address         */

/*------------------------------------------------------------------------
 * nulluser - initialize the system and become the null process
 *
 * Note: execution begins here after the C run-time environment has been
```

```
 * established.  Interrupts are initially DISABLED, and must eventually
 * be enabled explicitly.  The code turns itself into the null process
 * after initialization.  Because it must always remain ready to execute,
 * the null process cannot execute code that might cause it to be
 * suspended, wait for a semaphore, put to sleep, or exit.  In
 * particular, the code must not perform I/O except for polled versions
 * such as kprintf.
 *------------------------------------------------------------------------
 */

void    nulluser(void)
{
        kprintf("\n\r%s\n\n\r", VERSION);

        sysinit();

        /* Output Xinu memory layout */

        kprintf("%10d bytes physical memory.\r\n",
                (uint32)maxheap - (uint32)addressp2k(0));
        kprintf("           [0x%08X to 0x%08X]\r\n",
                (uint32)addressp2k(0), (uint32)maxheap - 1);
        kprintf("%10d bytes reserved system area.\r\n",
                (uint32)_start - (uint32)addressp2k(0));
        kprintf("           [0x%08X to 0x%08X]\r\n",
                (uint32)addressp2k(0), (uint32)_start - 1);
        kprintf("%10d bytes Xinu code.\r\n",
                (uint32)&_end - (uint32)_start);
        kprintf("           [0x%08X to 0x%08X]\r\n",
                (uint32)_start, (uint32)&_end - 1);
        kprintf("%10d bytes stack space.\r\n",
                (uint32)minheap - (uint32)&_end);
        kprintf("           [0x%08X to 0x%08X]\r\n",
                (uint32)&_end, (uint32)minheap - 1);
        kprintf("%10d bytes heap space.\r\n",
                (uint32)maxheap - (uint32)minheap);
        kprintf("           [0x%08X to 0x%08X]\r\n\r\n",
                (uint32)minheap, (uint32)maxheap - 1);

        /* Enable interrupts */

        enable();

        /* Create a process to execute function main() */
```

```
        resume(create
          ((void *)main, INITSTK, INITPRIO, "Main process", 0, NULL));

        /* Become the Null process (i.e., guarantee that the CPU has    */
        /*  something to run when no other process is ready to execute) */

        while (TRUE) {
                ;               /* do nothing */
        }
}

/*------------------------------------------------------------------------
 *
 * sysinit - intialize all Xinu data structures and devices
 *
 *------------------------------------------------------------------------
 */

static  void    sysinit(void)
{
        int32   i;
        struct  procent *prptr;         /* ptr to process table entry   */
        struct  dentry  *devptr;        /* ptr to device table entry    */
        struct  sentry  *semptr;        /* prr to semaphore table entry */
        struct  memblk  *memptr;        /* ptr to memory block          */

        /* Initialize system variables */

        /* Count the Null process as the first process in the system */

        prcount = 1;

        /* Scheduling is not currently blocked */

        Defer.ndefers = 0;

        /* Initialize the free memory list */

        maxheap = (void *)addressp2k(MAXADDR);

        memlist.mnext = (struct memblk *)minheap;

        /* Overlay memblk structure on free memory and set fields */

        memptr = (struct memblk *)minheap;
```

```
	memptr->mnext = NULL;
	memptr->mlength = memlist.mlength = (uint32)(maxheap - minheap);

	/* Initialize process table entries free */

	for (i = 0; i < NPROC; i++) {
		prptr = &proctab[i];
		prptr->prstate = PR_FREE;
		prptr->prname[0] = NULLCH;
		prptr->prstkbase = NULL;
		prptr->prprio = 0;
	}

	/* Initialize the Null process entry */

	prptr = &proctab[NULLPROC];
	prptr->prstate = PR_CURR;
	prptr->prprio = 0;
	strncpy(prptr->prname, "prnull", 7);
	prptr->prstkbase = minheap;
	prptr->prstklen = NULLSTK;
	prptr->prstkptr = 0;
	currpid = NULLPROC;

	/* Initialize semaphores */

	for (i = 0; i < NSEM; i++) {
		semptr = &semtab[i];
		semptr->sstate = S_FREE;
		semptr->scount = 0;
		semptr->squeue = newqueue();
	}

	/* Initialize buffer pools */

	bufinit();

	/* Create a ready list for processes */

	readylist = newqueue();

	/* Initialize real time clock */

	clkinit();
```

```
        /* Initialize non-volative RAM storage */

        nvramInit();

        for (i = 0; i < NDEVS; i++) {
                if (! isbaddev(i)) {
                        devptr = (struct dentry *) &devtab[i];
                        (devptr->dvinit) (devptr);
                }
        }
        return;
}
```

*Nulluser* itself is exceedingly simple. It calls procedure *sysinit* to initialize the system data structures. When *sysinit* returns, the running program has become the null process (process 0), but interrupts remain disabled and no other processes exist. After printing a few introductory messages, *nulluser* enables interrupts, and calls *create* to start a process running the user's main program.

Because the program executing *nulluser* has become the null process, it cannot exit, sleep, wait for a semaphore, or suspend itself. Fortunately, the initialization function does not perform any action that takes the caller out of the current or ready states. If such actions were needed, *sysinit* would have created another process to handle the action. Once initialization is complete and a process has been created to execute the user's main program, the null process falls into an infinite loop, giving *resched* a process to schedule when no user processes are ready to run.

## 22.7 Transforming A Program Into A Process

Function *sysinit* handles the task of system initialization. It initializes the system data structures, such as the semaphore table, the process table, and the free memory list. It also calls *clkinit* to initialize the real-time clock. Finally, *sysinit* iterates through the devices that have been configured and calls the initialization function for each device. To do so, it extracts field *dvinit* from the device switch table entry, treats the value as an address, and invokes the function at that address.

The most interesting piece of initialization code occurs about half-way through *sysinit* when it fills in the process table entry for process zero. Many of the process table fields, such as the process name field, can be left uninitialized — they are filled in merely to make debugging easier. The real work is done by the two lines that set the current process ID variable, *currpid*, to the ID of the null process, and assign *PR_CURR* to the process state field of the process table entry. Until *currpid* and the state have been assigned, rescheduling is impossible. Once they have been assigned, the program becomes a currently running process that *resched* can identify as the process with ID zero.

To summarize:

> *After it fills in the process table entry for process zero, the code sets variable* currpid *to zero, which transforms the sequential program into a process.*

Once the null process has been created, all that remains is for *sysinit* to initialize the other pieces of the system before it returns so that all services are available when function *nulluser* starts a process executing the user's main program.

## 22.8 Perspective

The intellectually significant aspects of operating system design arise when new abstractions are created on top of low-level hardware. In the case of system initialization, the details are not as important as the conceptual transformation: the processor starts a fetch-execute cycle running instructions sequentially, and the initialization code self-transforms into a concurrent processing system. The important point is that the initial code does not create a separate, concurrent system and then jump to the new system. No leap is made to reach the abstraction, and the original sequential execution is not abandoned. Instead, the running code declares itself to be a process, fills in the system data structures needed to support the declaration, and eventually allows other processes to execute. Meanwhile, the processor continues the fetch-execute cycle, and the new abstraction emerges without any disruption.

## 22.9 Summary

Initialization is the last step of system design; it should be postponed to avoid changing the design simply to make initialization easier. Although initialization involves many details, the most conceptually interesting part involves the transformation from a sequential program to a system that supports concurrent processing. To make itself correspond to the null process, the code fills in the process table entry for process zero and sets variable *currpid* to zero.

## EXERCISES

**22.1** If you were designing a bootstrap loader program, what additional functionality would you include? Why?

**22.2** Is the order of initialization important for the process table, semaphore table, memory free list, devices, and ready list? Explain.

**22.3** On many systems, it is possible to create a function *sizmem* that finds the highest valid memory address by probing memory until an exception occurs. Is such a function possible on the E2100L? Why or why not?

**22.4** Explain, by tracing through the functions involved, what would go wrong if *nulluser* enabled interrupts *before* calling *sysinit*.

**22.5** The network code, remote disk driver, and remote file system driver each create a process. Should the processes be created in *sysinit*? Why or why not?

**22.6** Most operating systems arrange for the network code to run and obtain an IP address before the system starts any user processes. Design a way for Xinu to create a network process, wait for the network process to obtain an IP address, and then create a process to run the main program? (Careful: the null process cannot block.)

# Chapter Contents

# 23

# Exception Handling

*I never make exceptions. An exception disproves the rule.*

— Sir Arthur Conan Doyle

## 23.1 Introduction

This chapter discusses the topic of exception handling. Because the underlying hardware dictates exactly how an exception is reported, the techniques an operating system uses to handle exceptions depend entirely on the hardware. Therefore, we will describe how exceptions are handled on the example E2100L, and leave readers to explore other systems and architectures.

In general, exception handling is concerned with details rather than concepts. Therefore, the chapter contains fewer broad concepts than earlier chapters. The reader should think of it as an example, and be aware that both the details and techniques may change if another hardware system is used.

## 23.2 Exceptions, Traps, And Illegal Interrupts

Arranging an operating system to correctly associate device addresses, interrupt vector locations, and interrupt dispatchers is a tedious task that plagues most implementers. Mismatches between the hardware and operating system configuration can cause devices to interrupt at vector locations other than those expected by the operating system. When a new device is added to an embedded system, the operating system software must be reconfigured to include a driver for the device.

A related problem occurs when bugs in a program cause it to generate an *exception*, sometimes called a *trap*. An exception might occur, for example, if a program attempts to fetch or store a word from memory using an address that is not a multiple of four or if a program attempts to divide by zero. Recall that when a program attempts an illegal instruction, the hardware handles the problem similar to the way it handles a device interrupt. That is, the hardware temporarily stops the fetch execute cycle, and uses the address found in an *exception vector* to pass control to the operating system function that handles exceptions.

When an unexpected interrupt occurs, the problem lies in the system configuration — the operating system must be reconfigured to handle the extra device. Exceptions are more complex. Exception handling depends on the size and purpose of the system. It also depends on whether the exception arose while the processor was executing operating system code (in which case the operating system vendor must correct the problem) or an application written by a third party (in which case the application vendor must correct the problem).

In a conventional system, when an application causes an exception, the operating system can terminate the process running the application and inform the user that a problem has occurred. For an embedded system, however, recovery is difficult or impossible. Even if the system allows interaction with a user, the user can do little to correct the problem. Thus, most embedded systems either reboot or power down.

Our example code follows the Unix tradition, and uses the name *panic* for the function that handles exceptions. Our version of *panic* takes a string as an argument. It displays the argument string on the console, and calls function *halt* to halt the processor. The code is minimal: *panic* does not attempt to recover, nor does it attempt to identify the offending process.

Because many hardware-specific details are involved, a version of *panic* that displays registers or processor state may need to be written in assembly language. For example, an exception can occur because a stack pointer is invalid, and a *panic* function may need to avoid using the stack. Consequently, to work under all circumstances, *panic* code cannot merely attempt to push a value on the stack or execute a function call. Similarly, because entries in the device switch table may be incorrect, a *panic* function that extracts information about the *CONSOLE* device from the device switch table may not work. Fortunately, most of these cases are extreme. Therefore, many operating system designers start with a basic version of the *panic* function that works as long as the bulk of the operating system and run-time environment remain intact.

## 23.3 Implementation Of Panic

Our version of a *panic* function is simplistic: it displays a message on the *CONSOLE*, turns off the "run" light (i.e., the front-panel LED), and calls *halt*. Because the MIPS architecture does not provide an instruction to stop the processor, our implementation of *halt* merely enters a tight loop. File *panic.c* contains the code:

```
/* panic.c - panic */

#include <xinu.h>

/*------------------------------------------------------------------------
 * panic - display a message and stop all processing
 *------------------------------------------------------------------------
 */
void    panic (
         char   *msg                    /* message to display           */
        )
{
        intmask mask;                   /* saved interrupt mask         */

        mask = disable();
        kprintf("\n\n\rpanic: %s\n\n\r", msg);
        gpioLEDOff(GPIO_LED_CISCOWHT);  /* turn off LED "run" light     */
        halt();                         /* halt the processor           */
}
```

## 23.4 Perspective

The question of how to handle exceptions is more complex than it may seem. To see why, consider what happens after an application makes a system call: although it remains running, the application process executes operating system code. If an exception occurs, the exception should be considered an operating system problem and should not invoke the exception handler for the process. Similarly, if an exception occurs while an application is executing code from a shared library, the exception should not be treated differently than an exception caused by the application. Such distinctions require the operating system to keep track of exactly what an application is currently doing.

Another type of complexity arises because exceptions can be caused by the interaction of processes. For example, if a Xinu process inadvertently writes into another process's address space, the second process may experience an exception that the first process caused. Thus, even if the system provides a mechanism to catch exceptions, the exception handler for the second process may not anticipate the problem and may have no way to recover.

## 23.5 Summary

Trapping and identifying exceptions and unexpected interrupts are important because they help isolate bugs that arise as an operating system is being implemented. Hence, building error detection functions early is essential, even if the implementation is crude and unsophisticated.

In embedded systems, an exception usually causes the system to reboot or power down. Our example implementation of *panic* assumes much of the operating system remains intact. The code disables interrupts, prints a message on the console, and calls *halt* to stop further processing.

## EXERCISES

**23.1** Rewrite Xinu to make the code serially reusable, and modify *panic* to wait 15 seconds and then jump back to the starting location (i.e., reboot Xinu).

**23.2** How many locations does *panic* require on the run-time stack to process an exception?

**23.3** Design a mechanism that allows an executing process to catch exceptions.

**23.4** An old LSI-11 computer included an exception for power-fail, and the authors once saw Xinu print a power-fail message. See if you can find a processor that detects imminent power failures, and find out how many instructions can be executed between the loss of power and system shutdown.

**23.5** Make a list of the requirements for the example implementation of *panic* to operate. (Hint: is a stack required?)

# NOTES

# Chapter Contents

# 24

# System Configuration

*No pleasure endures unseasoned by variety.*

— Publilius Syrus

## 24.1 Introduction

This chapter concludes the discussion of basic operating system design by answering a practical question: how can the code built in earlier chapters be transformed to make it suitable for a given computer that has a specific set of peripheral devices?

The chapter discusses the motivation for configuration, tradeoffs between static and dynamic configuration, and presents a basic configuration program that takes a description of the system and generates source files tailored to the description.

## 24.2 The Need For Multiple Configurations

The earliest computers were designed as monolithic systems. The hardware and software were designed together. A designer chose the details of the CPU, memory, and I/O devices, and an operating system was built to control the hardware. Later generations of computers added options, allowing a customer to choose between a large or small memory and a large or small disk. As the industry matured, third-party vendors began selling peripheral devices that could be attached to a computer. Current computers have many options — an owner can choose from among hardware devices sold by many vendors. Thus, a given computer can have a combination of hardware devices unlike other computers.

Designers use two approaches to accommodate combinations of devices:

- Static system configuration
- Dynamic device drivers

*Static system configuration*. Static configuration is used for small embedded systems that are "self-contained." For a typical embedded system, an operating system is created that supports the available hardware and does not contain any extra modules. To create such a system, the designer writes a specification that describes the exact peripherals on the system. The specification becomes input to a *configuration program* that manages operating system source code. The configuration program uses the specification to select the modules that are needed for the target hardware, and excludes modules for other hardware. When the resulting code has been compiled and linked, we say it is *configured* for the hardware.

*Dynamic device drivers*. Dynamic device drivers are used for large systems that have plenty of resources. The basic operating system starts running without an exact knowledge of the hardware. The system probes the hardware, determines which devices are present, and loads drivers for the devices automatically. Of course, the driver software must be available on a local disk, must be supplied by the device, or must be downloaded over the Internet. Furthermore, dynamic configuration takes time (i.e., extends the time required for bootstrap).

Static configuration is a form of early binding. The chief advantage is that the memory image only contains modules for the hardware that exists. Another advantage arises because the system does not spend time identifying hardware during the bootstrap process; the information is bound into the code. The chief disadvantage of early configuration is that a system configured for one machine cannot run on another unless the two are identical, including details such as the size of memory and all the devices.

Deferring configuration until system startup allows the designer to build more robust code because a single system can operate on several hardware configurations. During startup, the system can adapt itself to the hardware. More important, dynamic reconfiguration allows a system to adapt to changes in the hardware without stopping (e.g., when a user attaches or detaches a USB device).

## 24.3 Configuration In Xinu

Because it runs as an embedded system, Xinu follows a static configuration approach, with the majority of configuration occurring when the system is compiled and linked. Of course, even in some embedded systems, part of the configuration can be postponed until system startup. For example, some versions of Xinu calculate the size of memory and detect the presence of a real-time clock after Xinu begins. Interrupt vector initialization also occurs at system startup, when the system calls the driver initialization routines.

Xinu uses a configuration program to automate the selection of device driver modules. Named *config*, the program is not part of the operating system, and we do not need to examine the source code. Instead, we will look at how *config* operates: it takes an input file that contains specifications, and produces output files that become part of the operating system code. The next sections explain the configuration program and show examples.

## 24.4 Contents Of The Xinu Configuration File

The *config* program takes as input a text file named *Configuration*. It parses the input file and generates two output files, *conf.h* and *conf.c*. We have already seen the output files, which contain defined constants for devices and a definition of the device switch table.†

File *Configuration* is divided into three sections by occurrences of the separator characters "%%":

- Section 1: *type declarations* for device types
- Section 2: *device specifications* for specific devices
- Section 3: *automatically generated symbolic constants*

### 24.4.1 Section 1: Type Declarations

The motivation for a type declaration arises because a system may contain more than one copy of a particular hardware device. For example, if a system has two UART devices that both use the tty abstraction, the set of functions that comprise a tty driver must be specified for each UART. Entering the specification many times manually is error-prone, and can lead to inconsistencies. Thus, the type section allows the specification to be entered once and assigned a name that is used with both devices in the device specification section.

Each type declaration defines a name for the type, and lists a set of default device driver functions for the type. The declaration also allows one to specify the type of hardware with which the device is associated. For example, the type declaration:

```
tty:
        on uart
                -i ttyInit      -o ionull       -c ionull
                -r ttyRead      -g ttyGetc      -p ttyPutc
                -w ttyWrite     -s ioerr        -n ttyControl
                -intr ttyInterrupt              -irq 11
```

defines a type named *tty* that is used on a UART device. Neither *tty* nor *uart* is a keyword or has any meaning. Instead they are simply names that the designer chose. The

†File *conf.h* can be found on page 247, and *conf.c* can be found on page 259.

remaining items specify the default driver functions for type *tty*. Each driver function is preceded by a keyword that begins with a minus sign. Figure 24.1 lists the possible keywords and gives their meaning. Note: a given specification does not need to use all keywords.

| Keyword | Meaning |
|---|---|
| -i | function that performs init |
| -o | function that performs open |
| -c | function that performs close |
| -r | function that performs read |
| -w | function that performs write |
| -s | function that performs seek |
| -g | function that performs getc |
| -p | function that performs putc |
| -n | function that performs control |
| -intr | function that handles interrupts |
| -csr | control and status register address |
| -irq | interrupt vector number |

**Figure 24.1** Keywords used in the Xinu configuration file and their meaning.

### 24.4.2 Section 2: Device Specifications

Section 2 of file *Configuration* contains a declaration for each device in the system. A declaration gives a name for the device (e.g., *CONSOLE*), and specifies the set of functions that constitute a driver. Note that in Xinu, a device is an abstract concept, not necessarily tied to a physical hardware device. For example, in addition to devices like *CONSOLE* and *ETHERNET* that each correspond to underlying hardware, the device section can list *pseudo-devices*, such as a *FILE* device used for I/O.

Declaring a device serves two purposes. First, it allocates a slot in the device switch table, allowing the high-level I/O primitives to be used with the device rather than requiring a programmer to call specific driver functions. Second, it allows *config* to assign each device a minor device number. All devices with the same type are assigned minor numbers in sequence starting at zero.

When a device is declared, specific values can be supplied as needed or driver functions can be overridden. For example, the declaration:

```
CONSOLE is tty  on uart -csr 0xB8020000
```

declares *CONSOLE* to be a device of type *tty* that runs on UART hardware. In addition, the declaration specifies a CSR address of *0xB8020000*.

If a programmer wanted to test a new version of *ttyGetc*, the programmer might change the specification to:

```
CONSOLE is tty  on uart -csr 0xB8020000  -g myttyGetc
```

which uses the default driver functions from the tty declaration given above, but overrides the getc function and uses *myttyGetc*. Note that having a configuration makes it easy to change one function without modifying or replacing the original file.

The example type declaration includes the phrase *on uart*. To understand the purpose of specifying the underlying hardware, observe that designers sometimes wish to use the same abstraction for multiple pieces of hardware. For example, suppose a system contains two types of UART hardware. The *on* keyword allows a designer to use the tty abstraction for hardware types, and to allocate a single array of control blocks even though some of the low-level hardware details differ and the set of driver functions used for one type of hardware differs from the set used for another.

### 24.4.3 Automatically Generated Symbolic Constants

In addition to defining the structure of the device switch table, *conf.h* contains constants that specify the total number of devices and the number of each type. The *config* program generates the constants to reflect the specification found in file *Configuration*. For example, constant *NDEVS* is an integer that tells the total number of devices that have been configured. The device switch table contains *NDEVS* devices, and device-independent I/O routines use *NDEVS* to test whether a device id corresponds to a valid device.

*Config* also generates a set of defined constants that specify the number of devices of each type. Driver functions can use the appropriate constant to declare the array of control blocks. Each constant has the form N*xxx*, where *xxx* is the type name. For example, if file *Configuration* defines two devices of type *tty*, *conf.h* will contain the following line:

```
#define Ntty  2
```

## 24.5 Computation Of Minor Device Numbers

Consider the files that *config* produces. *Conf.h* contains the declaration of the device switch table, and *conf.c* contains the code that initializes the table. For a given device, its *devtab* entry contains a set of pointers to the device driver routines that correspond to high-level I/O operations like *open*, *close*, *read*, and *write*. The entry also contains the interrupt vector address, and the device's CSR address. All information in the device switch table is derived from file *Configuration* in a straightforward manner.

As mentioned above, each entry in the device switch table also contains a *minor device number*. Minor device numbers are nothing more than integers that distinguish among multiple devices that each use the same type of control block. Recall that device driver functions use the minor device number as an index into the array of control blocks to associate a specific entry with each device. In essence, the *config* program counts devices of each type. That is, each time it encounters a device, config uses the device type to assign the next minor device number (numbers start at zero). For example, Figure 24.2 shows how device IDs and minor numbers are assigned on a system that has three *tty* devices and two *eth* devices.

| device name | device identifier | device type | minor number |
|---|---|---|---|
| CONSOLE | 0 | tty | 0 |
| ETHERNET | 1 | eth | 0 |
| COM2 | 2 | tty | 1 |
| ETHER2 | 3 | eth | 1 |
| PRINTER | 4 | tty | 2 |

**Figure 24.2** An example of device configuration.

Notice that the three *tty* lines have minor numbers zero, one, and two, even though their device IDs happen to be zero, two, and four.

## 24.6 Steps In Configuring A Xinu System

To configure a Xinu system, the programmer edits file *Configuration*, adding or changing device information and symbolic constants as desired. When run, program *config* first reads and parses the file, collecting the information about each device type. It then reads device specifications, assigns minor device numbers, and produces the output files *conf.c* and *conf.h*. Finally, *config* appends definitions of symbolic constants from the third section of the specifications onto *conf.h*, making them available for operating system functions to include.

After *config* produces a new version of *conf.c* and *conf.h*, *conf.c* must be recompiled, as must all system functions that include *conf.h*.

## 24.7 Perspective

The history of operating systems is one of moving from static configuration to dynamic configuration. The interesting question is whether the benefits of dynamic configuration outweigh the costs. For example, compare booting Xinu to booting a large commercial operating system, such as Windows. Although the underlying computer hardware does not usually change, a commercial operating system always goes through the steps of polling the bus to find the devices that are present, loading drivers, and interacting with each device. If the operating system were only reconfigured when the hardware changed, the system could boot almost instantaneously, close to the boot time for Xinu. Now that you understand operating systems, you can think of the trade-off whenever you are forced to wait for an operating system to boot.

## 24.8 Summary

Instead of building a monolithic operating system tailored to specific hardware, designers look for ways to make systems configurable. Static configuration, a form of early binding, selects modules when the system is compiled and linked. The alternative, dynamic configuration, loads modules such as device drivers at run-time.

Because it is designed for an embedded environment, Xinu uses static configuration. Program *config* reads file *Configuration* and produces files *conf.h* and *conf.c* that define and initialize the device switch table. The separation of device types from device declarations allows *config* to compute minor device numbers.

## EXERCISES

**24.1** Create a function *myttyRead* that calls *ttyGetc* repeatedly to satisfy a request. To test your code, modify file *Configuration* to substitute your code in place of *ttyRead*.

**24.2** Find out how other systems are configured. What happens, for example, when Windows boots?

**24.3** If every operating system function includes *conf.h*, any change to file *Configuration* means a new version of *conf.h* will be generated, and the entire system must be recompiled. Redesign the *config* program that separates constants into several different include files to eliminate unnecessary recompilation.

**24.4** Discuss whether a configuration program is worthwhile. Include some estimate of the extra effort required to make a system easily configurable. Remember that a programmer is likely to have little experience or knowledge about a system when it is first configured.

**24.5** In theory, many aspects of a system may need to change when porting the system from one computer to another. In addition to devices, for example, one might consider the processor (not only the basic instruction set, but the extra instructions found on some models), the availability of co-processors (including floating point), the real-time clock or time resolution, and the endianness of integers. Argue that if a configuration system has parameters for all the above, the resulting system is untestable.

# Chapter Contents

# 25

# An Example User Interface: The Xinu Shell

*A man has to learn that he cannot command things...*

— James Allen

## 25.1 Introduction

Previous chapters explain an operating system as a set of functions that applications can invoke to obtain services. However, a typical user never encounters system functions. Instead, users invoke applications and allow the applications to access functions in the underlying system.

This chapter examines a basic interface known as a *shell*† that allows a user to launch applications and control their input and output. Following the pattern established by other parts of the system, our design emphasizes simplicity and elegance rather than features. We concentrate on a few fundamental ideas that make the shell powerful without requiring large amounts of code. The chapter shows examples of both software that interprets user commands and applications that a user can invoke. Although it only offers basic functionality, our example interpreter illustrates several important concepts.

†The term *shell* and many of the ideas used in the Xinu shell come from UNIX.

## 25.2 What Is A User Interface?

A *user interface* consists of the hardware and software with which users interact to perform computational tasks and observe the results. Thus, user interface software lies between a human who specifies what must be done and a computer system that performs the specified tasks.

The goal of user interface design is to create an environment in which users can perform computational tasks conveniently and productively. For example, most modern user interfaces employ a graphical representation that presents a set of icons from which the user selects to launch an application. The use of graphics makes application selection quick, and relieves the user from memorizing a set of application names.

Small embedded systems typically offer two levels of user interface: one for the end user and another for a system builder. The E2100L, our example system, provides both levels: a web interface for customers and a console interface for programmers. When a customer uses the wireless router as the vendor intended, the customer boots the router and then uses a conventional browser to interact with the device. The web interface allows a customer to set passwords, control the wireless network hardware, and configure routing. However, the web interface cannot be used to download software or otherwise change the system. To perform such tasks, a programmer must use the console interface and interact over a serial line.

## 25.3 Commands And Design Principles

Industry uses the term *Command Line Interface* (*CLI*) to describe a user interface that allows a user to enter a series of textual commands; many embedded system products offer a command-line interface. Usually, each line of input corresponds to a command, and the system processes a line before reading the next line. The term *command* arises because most CLIs follow the same syntactic form in which a line starts with a name that specifies an action to be taken and is followed by parameters that control the details of the action and the item(s) to which the action applies. For example, if we can imagine a system that uses a command named *config* to control settings associated with a network interface, the command to set the MTU parameter of interface number 0 to 1500 might be:

*config 0 MTU=1500*

The set of all available commands determines the functionality available to a user (i.e., defines the power of the computing system). However, a good design does not merely collect random commands. Instead, the design adheres to the following principles:

- Functionality: sufficient for all needs
- Orthogonality: only one way to perform a given task
- Consistency: commands follow a consistent pattern
- Least astonishment: a user should be able to predict results

## 25.4 Design Decisions For A Simplified Shell

When a programmer designs a shell, the programmer must choose among many alternatives. The following paragraphs list decisions a programmer faces, and describe the choices made for a simplified Xinu shell.

*Handling input*. Should our interface allow the terminal device driver to handle the details of backspacing, character echoing, and line erasing, or should it handle those details itself? The choice is important because it determines the extent to which the shell can control input. For example, a modern Unix shell allows *Control-B* and *Control-F* to move the cursor backward and forward while editing a line of input.† However, the Xinu tty driver does not provide such editing features. Despite its limitations, our example shell uses the tty driver to simplify the design and reduce the amount of code.

*Foreground or background execution*. Does the shell wait while a command completes execution before starting another? Our shell follows the Unix tradition of allowing a user to decide whether the shell waits or the command executes in background.

*Control of input and output*. Also like Unix, our example shell allows a user to specify a source of input and destination for output when the command is invoked. The technique, known as *I/O redirection*, allows each command to function as a general-purpose tool that can be applied to a variety of files and I/O devices. Providing redirection in the shell also means that I/O specifications are uniform — a single mechanism for redirection applies to all commands.

*Typed or untyped arguments*. The question is whether the shell understands the number and type of arguments for a given command. Following the Unix tradition, our example shell does not understand arguments, and does not interpret them. Instead, the shell treats each argument as a text string, and the set of arguments is passed to the command. Consequently, each command must check whether its arguments are valid.

## 25.5 Shell Organization And Operation

A shell is organized as a loop that repeatedly reads a line of input and executes the command on the line. Once a line has been read, the shell must extract a command name, arguments, and other items, such as the specifications of I/O redirection or background processing. Following standard practice for syntactic analysis, we will divide the code into two functions: one that handles lexical analysis by grouping characters into *tokens*, and another that examines whether the set of tokens form a valid command.

Using a separate lexical function may seem unnecessary for the trivial syntax of our sample shell. However, we chose the organization because it permits future expansion.

†The use of *Control-B* and *Control-F* is derived from the Emacs editor.

## 25.6 The Definition Of Lexical Tokens

At the lexical level, our shell scans a line of input and groups characters into syntactic tokens. Figure 25.1 lists the lexical tokens that the scanner recognizes and the four lexical types the scanner uses when classifying tokens.

| Token Type (num. value) | Character | Description |
|---|---|---|
| SH_TOK_AMPER (0) | & | ampersand |
| SH_TOK_LESS (1) | < | less-than symbol |
| SH_TOK_GREATER (2) | > | greater-than symbol |
| SH_TOK_OTHER (3) | '...' | quoted string (single quotes) |
| SH_TOK_OTHER (3) | "..." | quoted string (double-quotes) |
| SH_TOK_OTHER (3) | other | sequence of non-whitespace |

**Figure 25.1** Lexical tokens used by the example Xinu shell.

The use of quoted strings allows a user to specify an argument (or a file name) that contains arbitrary characters, including the special characters recognized by the shell. Each quoted string starts with either a single or double quote, and contains all characters, including blanks and tabs, up to the first occurrence of the opening quote. Thus, the string:

```
'a string'
```

contains eight total characters including a blank, and the string

```
"don't blink"
```

contains eleven characters, including a single quote. The lexical scanner removes the surrounding quotes, and classifies the resulting sequence of characters to be a single token of type *SH_TOK_OTHER*.

The lexical scanner defines *whitespace* to consist of blanks or tab characters. At least one whitespace character must separate two tokens of type *SH_TOK_OTHER*. Otherwise, whitespace is ignored.

## 25.7 The Definition Of Command-Line Syntax

Once a line has been scanned and divided into a series of lexical tokens, the shell parses the tokens to verify that they form a valid sequence. The syntax is:

command_name args* [ redirection ] [ background ]

Brackets [ ] denote an optional occurrence and asterisk indicates zero or more repetitions of an item. The string *command_name* denotes the name of a command, *args** denotes zero or more optional arguments, the optional *redirection* refers to input redirection, output redirection, or both, and the optional *background* indicates background execution. Figure 25.2 contains a grammar that defines the set of valid inputs in terms of tokens.

| | | |
|---|---|---|
| **command** | → | **name [args] [redirection] [background]** |
| **name** | → | **SH_TOK_OTHER** |
| **args** | → | **SH_TOK_OTHER [args]** |
| **redirection** | → | **input_redirect [output_redirect]** |
| **redirection** | → | **output_redirect [input_redirect]** |
| **input_redirect** | → | **SH_TOK_LESS SH_TOK_OTHER** |
| **output_redirect** | → | **SH_TOK_GREATER SH_TOK_OTHER** |
| **background** | → | **SH_TOK_AMPER** |

**Figure 25.2** A grammar that specifies valid sequences of tokens for the example shell.

In essence, a command consists of a sequence of one or more "other" tokens optionally followed by requests to redirect the input and/or output (in either order), optionally followed by a specification that the command should run in background. The first token on the line must be the name of a command.

## 25.8 Implementation Of The Xinu Shell

Our examination of the implementation begins with the definition of constants and variables used by the shell. File *shell.h* contains the declarations.

```
/* shell.h - declarations and constants used by the Xinu shell */

/* Size constants */

#define SHELL_BUFLEN    TY_IBUFLEN+1    /* length of input buffer       */
#define SHELL_MAXTOK    32              /* maximum tokens per line      */
#define SHELL_CMDSTK    8192            /* size of stack for process    */
                                        /*    that executes command     */
#define SHELL_ARGLEN    (SHELL_BUFLEN+SHELL_MAXTOK) /* argument area    */
#define SHELL_CMDPRIO   20              /* process priority for command */

/* Message constants */

/* Shell banner (assumes VT100) */

#define SHELL_BAN0      "\033[1;31m"
#define SHELL_BAN1      "-----------------------------------------"
#define SHELL_BAN2      "   __    __   _____    _   _    _    _    "
#define SHELL_BAN3      "   \\ \\  / /  |__ __|  | \\ | |  | |  | |   "
#define SHELL_BAN4      "    \\ \\/ /     | |    |  \\| |  | |  | |   "
#define SHELL_BAN5      "    / /\\ \\    _| |_   | \\   |  | |  | |   "
#define SHELL_BAN6      "   / /  \\ \\  |     |  | | \\ |  \\  --  /   "
#define SHELL_BAN7      "   --    --   -----    -   -     ----     "
#define SHELL_BAN8      "-----------------------------------------"
#define SHELL_BAN9      "\033[0;39m\n"

/* Messages shell displays for user */

#define SHELL_PROMPT    "xsh $ "                /* prompt               */
#define SHELL_STRTMSG   "Welcome to Xinu!\n"    /* Welcome message      */
#define SHELL_EXITMSG   "Shell closed\n"        /* shell exit message   */
#define SHELL_SYNERRMSG "Syntax error\n"        /* syntax error message */
#define SHELL_CREATMSG  "Cannot create process\n"/* command error       */
#define SHELL_INERRMSG  "Cannot open file %s for input\n" /* input err  */
#define SHELL_OUTERRMSG "Cannot open file %s for output\n"/* output err */
#define SHELL_BGERRMSG  "Cannot redirect I/O or background a builtin\n"
                                                /* builtin cmd err      */
/* Constants used for lexical analysis */

#define SH_NEWLINE      '\n'            /* New line character           */
#define SH_EOF          '\04'           /* Control-D is EOF             */
#define SH_AMPER        '&'             /* ampersand character          */
#define SH_BLANK        ' '             /* blank character              */
#define SH_TAB          '\t'            /* tab character                */
#define SH_SQUOTE       '\''            /* single quote character       */
```

```
#define SH_DQUOTE       '"'             /* double quote character       */
#define SH_LESS         '<'             /* less-than character  */
#define SH_GREATER      '>'             /* greater-than character       */

/* Token types */

#define SH_TOK_AMPER    0               /* ampersand token              */
#define SH_TOK_LESS     1               /* less-than token              */
#define SH_TOK_GREATER  2               /* greater-than token           */
#define SH_TOK_OTHER    3               /* token other than those       */
                                        /*   listed above (e.g., an     */
                                        /*   alphanumeric string)       */

/* Shell return constants */

#define SHELL_OK         0
#define SHELL_ERROR      1
#define SHELL_EXIT      -3

/* Structure of an entry in the table of shell commands */

struct  cmdent  {                       /* entry in command table       */
        char    *cname;                 /* name of command              */
        bool8   cbuiltin;               /* is this a builtin command?   */
        int32   (*cfunc)(int32,char*[]);/* function for command         */
};

extern  uint32  ncmd;
extern  const   struct  cmdent  cmdtab[];
```

The final section of file *shell.h* defines table *cmdtab* that holds information about shell commands. Each entry in the table is struct *cmdent* which contains three items: a name for the command, a Boolean indicating whether the command is restricted to run as a builtin, and a pointer to a function that implements the command. Later sections discuss how the command table is initialized and how it is used.

## 25.9 Storage Of Tokens

The data structures that our shell uses are somewhat unexpected. An integer array, *toktyp*, is used to record the type of each token. The tokens are stored as null-terminated strings, packed in contiguous locations of character array *tokbuf*. An integer array, *tok*, is used to store the index of the start of each token. The shell depends on two counters: *ntok* counts the number of tokens found so far, and variable *tlen* counts

the characters that have been stored in array *tokbuf*. To understand the data structures, consider the example input line:

date > file &

The line contains four tokens. Figure 25.3 shows how the lexical analyzer fills in the data structures to hold the tokens that have been extracted from the input line.

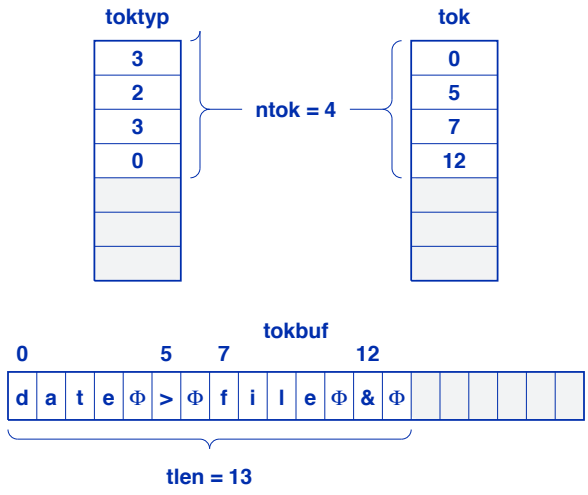



**Figure 25.3** Contents of variables *tokbuf*, *toktyp*, *tok*, *ntok*, and *tlen* for the input line: *date > file &*

As the figure shows, the tokens themselves are placed in array *tokbuf* with all whitespace removed. A null character terminates each token. Array *tok* contains integers that are each an index into *tokbuf* — the $i^{th}$ location of array *tok* gives the index in *tokbuf* of the string for the $i^{th}$ token. Finally, the $i^{th}$ location of array *toktyp* specifies the type of the $i^{th}$ token. For example, the third token on the line, *file*, has type 3 (*SH_TOK_OTHER*†), and starts at location 7 in array *tokbuf*.

## 25.10 Code For The Lexical Analyzer

Because our shell syntax is straightforward, we have chosen to use an ad hoc implementation for the lexical analyzer. File *lexan.c* contains the code.

†Figure 25.1 on page 552 lists the numeric values for each token type.

```
/* lexan.c  - lexan */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lexan - ad hoc lexical analyzer to divide command line into tokens
 *------------------------------------------------------------------------
 */

int32   lexan (
          char          *line,          /* input line terminated with   */
                                        /*  NEWLINE or NULLCH           */
          int32         len,            /* length of the input line,    */
                                        /*   including NEWLINE          */
          char          *tokbuf,        /* buffer into which tokens are */
                                        /*   stored with a null         */
                                        /*   following each token       */
          int32         *tlen,          /* place to store number of     */
                                        /*   chars in tokbuf            */
          int32         tok[],          /* array of pointers to the     */
                                        /*   start of each token        */
          int32         toktyp[]        /* array that gives the type    */
                                        /*   of each token              */
        )
{
        char    quote;                  /* character for quoted string  */
        uint32  ntok;                   /* number of tokens found       */
        char    *p;                     /* pointer that walks along the */
                                        /*   input line                 */
        int32   tbindex;                /* index into tokbuf            */
        char    ch;                     /* next char from input line    */

        /* Start at the beginning of the line with no tokens */

        ntok = 0;
        p = line;
        tbindex = 0;

        /* While not yet at end of line, get next token */

        while ( (*p != NULLCH) && (*p != SH_NEWLINE) ) {

                /* If too many tokens, return error */

                if (ntok >= SHELL_MAXTOK) {
```

```
        return SYSERR;
}

/* Skip whitespace before token */

while ( (*p == SH_BLANK) || (*p == SH_TAB) ) {
        p++;
}

/* Stop parsing at end of line (or end of string) */

ch = *p;
if ( (ch==SH_NEWLINE) || (ch==NULLCH) ) {
        *tlen = tbindex;
        return ntok;
}

/* Set next entry in tok array to be an index to the    */
/*   current location in the token buffer               */

tok[ntok] = tbindex;    /* the start of the token       */

/* Set the token type */

switch (ch) {

    case SH_AMPER:      toktyp[ntok] = SH_TOK_AMPER;
                        tokbuf[tbindex++] = ch;
                        tokbuf[tbindex++] = NULLCH;
                        ntok++;
                        p++;
                        continue;

    case SH_LESS:       toktyp[ntok] = SH_TOK_LESS;
                        tokbuf[tbindex++] = ch;
                        tokbuf[tbindex++] = NULLCH;
                        ntok++;
                        p++;
                        continue;

    case SH_GREATER:    toktyp[ntok] = SH_TOK_GREATER;
                        tokbuf[tbindex++] = ch;
                        tokbuf[tbindex++] = NULLCH;
                        ntok++;
                        p++;
```

```
                                continue;

            default:            toktyp[ntok] = SH_TOK_OTHER;
        };

        /* Handle quoted string (single or double quote) */

        if ( (ch==SH_SQUOTE) || (ch==SH_DQUOTE) ) {
                quote = ch;     /* remember opening quote */

                /* Copy quoted string to arg area */

                p++;    /* Move past starting quote */

                while ( ((ch = *p++) != quote) && (ch != SH_NEWLINE)
                                && (ch != NULLCH) ) {
                        tokbuf[tbindex++] = ch;
                }
                if (ch != quote) {  /* string missing end quote */
                        return SYSERR;
                }

                /* Finished string - count token and go on      */

                tokbuf[tbindex++] = NULLCH; /* terminate token  */
                ntok++;         /* count string as one token    */
                continue;       /* go to next token             */
        }

        /* Handle a token other than a quoted string           */

        tokbuf[tbindex++] = ch; /* put first character in buffer*/
        p++;

        while ( ((ch = *p) != SH_NEWLINE) && (ch != NULLCH)
                && (ch != SH_LESS)  && (ch != SH_GREATER)
                && (ch != SH_BLANK) && (ch != SH_TAB)
                && (ch != SH_AMPER) && (ch != SH_SQUOTE)
                && (ch != SH_DQUOTE) )  {
                        tokbuf[tbindex++] = ch;
                        p++;
        }

        /* Report error if other token is appended */
```

```
                if (       (ch == SH_SQUOTE) || (ch == SH_DQUOTE)
                        || (ch == SH_LESS)   || (ch == SH_GREATER) ) {
                        return SYSERR;
                }

                tokbuf[tbindex++] = NULLCH;     /* terminate the token  */

                ntok++;                         /* count valid token    */

        }
        *tlen = tbindex;
        return ntok;
}
```

The first two arguments of *lexan* give the address of an input line and the length of the line. Succeeding arguments give pointers to the data structures that Figure 25.3 illustrates. *Lexan* initializes the number of tokens found, a pointer to the input line, and an index into array tokbuf. It then enters a *while* loop that runs until pointer *p* reaches the end of the file.

To process a token, *lexan* skips leading white space (i.e., blanks and tabs), stores the current index of *tokbuf* in array *tok*, and then uses a switch statement to choose the action appropriate for the next input character. For any of the three single-character tokens (i.e., an ampersand, less-than symbol, or greater-than symbol), *lexan* records the token type in array *toktyp*, places the token followed by a null character in the *tokbuf* array, increments *ntok*, moves to the next character in the string, and continues the while loop, which will start to process the next input character.

For a character that is not one of the three single-character tokens, *lexan* records the token type as *SH_TOK_OTHER* and falls through the switch statement. There are two cases: the token is a quoted string or the token consists of contiguous characters up to the next special character or whitespace. *Lexan* recognizes either a single-quote or a double-quote character; the string ends at the first occurrence of a matching quote or the end-of-line, whichever occurs first. If it encounters an end-of-line condition, *lexan* returns *SYSERR*. Otherwise, it copies characters from the string to *tokbuf* unchanged and uninterpreted, which means that a string can contain arbitrary characters, including whitespace and the other quote mark character. Once the copy has been completed, *lexan* appends a null character to define the end of the token. It then continues the outer while loop to look for the next token.

The final section of code handles a token that is composed of contiguous characters other than the single token characters and quotes. The code loops until it encounters a special character or whitespace, placing characters in successive locations of *tokbuf*. Before moving on to the next token, the code checks for an error where two tokens occur with no whitespace between them.

Once *lexan* reaches the end of the input line, it returns a count of the number of tokens found. If an error is detected during processing, *lexan* returns *SYSERR* to its caller, making no attempt to recover or repair the problem. That is, the action to be taken when an error occurs is coded into *lexan*. The exercises discuss the choice of error handling and suggest alternatives.

## 25.11 The Heart Of The Command Interpreter

Although a command interpreter must handle many details, the basic algorithm is not difficult to understand. In essence, the code consists of a loop that repeatedly reads a line of input, uses *lexan* to extract tokens, checks the syntax, arranges a way to pass arguments, redirects I/O if necessary, and runs a command in foreground or background as specified. The loop terminates if a user enters the end-of-file character (control-d), or if a command returns a special exit code.

As with the lexical analyzer, our interpreter implementation uses an ad hoc implementation. That is, the code does not resemble a conventional compiler, nor does it contain independent code to verify that the sequence of tokens is valid. Instead, error checking is built into each step of processing. For example, after it has processed background and I/O redirection, the shell verifies that remaining tokens are all of type *SH_TOK_OTHER*.

Examining the code will make the approach clear. Function *shell* performs command interpretation; file *shell.c*, shown below, contains the code. Note that the file also includes the declaration of array *cmdtab* which specifies the set of commands and the function used to implement each. The code also sets external variable *ncmd* to the number of commands in the table.

Conceptually, the set of commands is independent from the code that processes user input. Thus, it may make sense to divide *shell.c* into two files: one that specifies commands and another that contains code. In practice, however, the two have been combined because the example set of commands is so small that an additional file is unnecessary.

```
/* shell.c  -  shell */

#include <xinu.h>
#include <stdio.h>
#include "shprototypes.h"

/************************************************************************/
/* Xinu shell commands and the function associated with each            */
/************************************************************************/
const   struct  cmdent  cmdtab[] = {
        {"argecho",     TRUE,   xsh_argecho},
        {"arp",         FALSE,  xsh_arp},
        {"cat",         FALSE,  xsh_cat},
        {"clear",       TRUE,   xsh_clear},
        {"date",        FALSE,  xsh_date},
        {"devdump",     FALSE,  xsh_devdump},
        {"echo",        FALSE,  xsh_echo},
        {"ethstat",     FALSE,  xsh_ethstat},
        {"exit",        TRUE,   xsh_exit},
        {"help",        FALSE,  xsh_help},
        {"ipaddr",      FALSE,  xsh_ipaddr},
        {"kill",        TRUE,   xsh_kill},
        {"led",         FALSE,  xsh_led},
        {"memdump",     FALSE,  xsh_memdump},
        {"memstat",     FALSE,  xsh_memstat},
        {"nvram",       FALSE,  xsh_nvram},
        {"ping",        FALSE,  xsh_ping},
        {"ps",          FALSE,  xsh_ps},
        {"sleep",       FALSE,  xsh_sleep},
        {"udpdump",     FALSE,  xsh_udpdump},
        {"udpecho",     FALSE,  xsh_udpecho},
        {"udpeserver",  FALSE,  xsh_udpeserver},
        {"uptime",      FALSE,  xsh_uptime},
        {"?",           FALSE,  xsh_help}

};

uint32  ncmd = sizeof(cmdtab) / sizeof(struct cmdent);

/************************************************************************/
/* Xinu shell - provide an interactive user interface that executes     */
/*              commands.  Each command begins with a command name, has */
/*              a set of optional arguments, has optional input or      */
/*              output redirection, and an optional specification for   */
/*              background execution (ampersand).  The syntax is:       */
```

```
/*                                                                     */
/*                  command_name [args*] [redirection] [&]             */
/*                                                                     */
/*              Redirection is either or both of:                      */
/*                                                                     */
/*                              < input_file                           */
/*                      or                                             */
/*                              > output_file                          */
/*                                                                     */
/***********************************************************************/

process shell (
                did32   dev             /* ID of tty device from which  */
        )                               /*  to accept commands          */
{
        char    buf[SHELL_BUFLEN];      /* input line (large enough for */
                                        /*  one line from a tty device  */
        int32   len;                    /* length of line read          */
        char    tokbuf[SHELL_BUFLEN +   /* buffer to hold a set of      */
                        SHELL_MAXTOK];  /* contiguous null-terminated   */
                                        /* strings of tokens            */
        int32   tlen;                   /* current length of all data   */
                                        /*   in array tokbuf            */
        int32   tok[SHELL_MAXTOK];      /* index of each token in       */
                                        /*   tokbuf                     */
        int32   toktyp[SHELL_MAXTOK];   /* type of each token in tokbuf */
        int32   ntok;                   /* number of tokens on line     */
        pid32   child;                  /* process ID of spawned child  */
        bool8   backgnd;                /* run command in background?   */
        char    *outname, *inname;      /* ptrs to strings for file     */
                                        /*   names that follow > and <  */
        did32   stdinput, stdoutput;    /* descriptors for redirected   */
                                        /*   input and output           */
        int32   i;                      /* index into array of tokens   */
        int32   j;                      /* index into array of commands */
        int32   msg;                    /* message from receive() for   */
                                        /*   child termination          */
        int32   tmparg;                 /* address of this var is used  */
                                        /*   when first creating child  */
                                        /*   process, but is replaced   */
        char    *src, *cmp;             /* ptrs using during name       */
                                        /*   comparison                 */
        bool8   diff;                   /* was difference found during  */
                                        /*   comparison                 */
        char    *args[SHELL_MAXTOK];    /* argument vector passed to    */
                                        /*   builtin commands           */
```

```
        /* Print shell banner and startup message */

        fprintf(dev, "\n\n%s%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
                SHELL_BAN0,SHELL_BAN1,SHELL_BAN2,SHELL_BAN3,SHELL_BAN4,
                SHELL_BAN5,SHELL_BAN6,SHELL_BAN7,SHELL_BAN8,SHELL_BAN9);

        fprintf(dev, "%s\n\n", SHELL_STRTMSG);

        /* Continually prompt the user, read input, and execute command */

        while (TRUE) {

                /* Display prompt */

                fprintf(dev, SHELL_PROMPT);

                /* Read a command (for tty, 0 means entire line) */

                len = read(dev, buf, sizeof(buf));

                /* Exit gracefully on end-of-file */

                if (len == EOF) {
                        break;
                }

                /* If line contains only NEWLINE, go to next line */

                if (len <= 1) {
                        fprintf(dev,"\n");
                        continue;
                }

                buf[len] = SH_NEWLINE;  /* terminate line */

                /* Parse input line and divide into tokens */

                ntok = lexan(buf, len, tokbuf, &tlen, tok, toktyp);

                /* Handle parsing error */

                if (ntok == SYSERR) {
                        fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                        continue;
                }
```

```
/* If line is empty, go to next input line */

if (ntok == 0) {
        fprintf(dev, "\n");
        continue;
}

/* If last token is '&', set background */

if (toktyp[ntok-1] == SH_TOK_AMPER) {
        ntok-- ;
        tlen-= 2;
        backgnd = TRUE;
} else {
        backgnd = FALSE;
}


/* Check for input/output redirection (default is none) */

outname = inname = NULL;
if ( (ntok >=3) && ( (toktyp[ntok-2] == SH_TOK_LESS)
                    ||(toktyp[ntok-2] == SH_TOK_GREATER))) {
        if (toktyp[ntok-1] != SH_TOK_OTHER) {
                fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                continue;
        }
        if (toktyp[ntok-2] == SH_TOK_LESS) {
                inname =  &tokbuf[tok[ntok-1]];
        } else {
                outname = &tokbuf[tok[ntok-1]];
        }
        ntok -= 2;
        tlen = tok[ntok] - 1;
}


if ( (ntok >=3) && ( (toktyp[ntok-2] == SH_TOK_LESS)
                    ||(toktyp[ntok-2] == SH_TOK_GREATER))) {
        if (toktyp[ntok-1] != SH_TOK_OTHER) {
                fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                continue;
        }
        if (toktyp[ntok-2] == SH_TOK_LESS) {
                if (inname != NULL) {
```

```
                        fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                        continue;
                    }
                    inname = &tokbuf[tok[ntok-1]];
            } else {
                    if (outname != NULL) {
                        fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                        continue;
                    }
                    outname = &tokbuf[tok[ntok-1]];
            }
            ntok -= 2;
            tlen = tok[ntok] - 1;
    }

    /* Verify remaining tokens are type "other" */

    for (i=0; i<ntok; i++) {
            if (toktyp[i] != SH_TOK_OTHER) {
                    break;
            }
    }
    if ((ntok == 0) || (i < ntok)) {
            fprintf(dev, SHELL_SYNERRMSG);
            continue;
    }

    stdinput = stdoutput = dev;

    /* Lookup first token in the command table */

    for (j = 0; j < ncmd; j++) {
            src = cmdtab[j].cname;
            cmp = tokbuf;
            diff = FALSE;
            while (*src != NULLCH) {
                    if (*cmp != *src) {
                            diff = TRUE;
                            break;
                    }
                    src++;
                    cmp++;
            }
            if (diff) {
                    continue;
```

```
                } else {
                        break;
                }
        }

        /* Handle command not found */

        if (j >= ncmd) {
                fprintf(dev, "command %s not found\n", tokbuf);
                continue;
        }

        /* Handle built-in command */

        if (cmdtab[j].cbuiltin) { /* no background or redirection */
                if (inname != NULL || outname != NULL || backgnd) {
                        fprintf(dev, SHELL_BGERRMSG);
                        continue;
                } else {
                        /* Set up arg vector for call */

                        for (i=0; i<ntok; i++) {
                                args[i] = &tokbuf[tok[i]];
                        }

                        /* Call builtin shell function */

                        if ((*cmdtab[j].cfunc)(ntok, args)
                                                == SHELL_EXIT) {
                                break;
                        }
                }
                continue;
        }

        /* Open files and redirect I/O if specified */

        if (inname != NULL) {
                stdinput = open(NAMESPACE,inname,"ro");
                if (stdinput == SYSERR) {
                        fprintf(dev, SHELL_INERRMSG, inname);
                        continue;
                }
        }
        if (outname != NULL) {
```

```
                        stdoutput = open(NAMESPACE,outname,"w");
                        if (stdoutput == SYSERR) {
                                fprintf(dev, SHELL_OUTERRMSG, outname);
                                continue;
                        } else {
                                control(stdoutput, F_CTL_TRUNC, 0, 0);
                        }
                }

                /* Spawn child thread for non-built-in commands */

                child = create(cmdtab[j].cfunc,
                        SHELL_CMDSTK, SHELL_CMDPRIO,
                        cmdtab[j].cname, 2, ntok, &tmparg);

                /* If creation or argument copy fails, report error */

                if ((child == SYSERR) ||
                    (addargs(child, ntok, tok, tlen, tokbuf, &tmparg)
                                                        == SYSERR) ) {
                        fprintf(dev, SHELL_CREATMSG);
                        continue;
                }

                /* Set stdinput and stdoutput in child to redirect I/O */

                proctab[child].prdesc[0] = stdinput;
                proctab[child].prdesc[1] = stdoutput;

                msg = recvclr();
                resume(child);
                if (! backgnd) {
                        msg = receive();
                        while (msg != child) {
                                msg = receive();
                        }
                }
        }

        /* Close shell */

        fprintf(dev,SHELL_EXITMSG);
        return OK;
}
```

The main loop calls *lexan* to divide the input line into tokens, and begins processing the command. First, the code checks the last token to see if the user appended an ampersand. If so, the shell sets Boolean *backgnd* to *TRUE*; otherwise, *backgnd* is set to *FALSE*. The variable is used later to determine whether to run the command in background.

After the background token has been removed, the shell checks for I/O redirection. Input and output redirection can both be specified, and the specifications can occur in either order, but must be the last of the remaining tokens. Therefore, the shell checks for redirection twice. If two specifications occur, the shell verifies that they do not both specify input or both specify output. At this point in processing the line, the shell merely saves a pointer to the file name without attempting to open the file (the files are opened later).

Once the shell has removed tokens that specify I/O redirection, the only tokens that remain correspond to a command name and arguments to the command. Thus, before it continues to process the command, the shell iterates through remaining tokens to verify that they are of type "other" (*SH_TOK_OTHER*). If any are not, the code prints an error message and moves to the next input line. Once all checks have been performed, the shell looks up the command and executes the corresponding function.

## 25.12 Command Name Lookup And Builtin Processing

The first token on the line is a command name. Recall that the example code stores information about commands in array *cmdtab*. Thus, lookup is straightforward — the shell searches the array sequentially looking for an exact match between the first token and one of the command names. If no match is found, the code prints an error message and moves to the next command.

Our shell supports two types of commands: builtin and non-builtin. The difference arises from the way the commands are executed: the shell uses the conventional function call mechanism to execute a builtin command, and creates a separate process to execute a non-builtin command. The distinction means that a user cannot specify background processing and cannot redirect I/O for a builtin command.†

To test whether a command should execute as a builtin, the shell examines field *cbuiltin* of the entry in *cmdtab*. For a builtin command, no redirection or background processing is allowed. So, the shell verifies that neither was specified, creates an argument array in variable *args*, and calls the command function. The next section explains how command arguments are constructed.

†An exercise suggests a way to blur the distinction between builtin and non-builtin commands.

## 25.13 Arguments Passed To Commands

Our example shell uses the same argument passing mechanism as a Unix shell. When a command is invoked, the shell passes tokens from the command line as uninterpreted, null-terminated strings. The shell does not know how many arguments a given command expects, nor does the shell understand whether the arguments make sense. Instead, the shell merely passes all arguments from the command line, and allows the command to check and interpret them.

Conceptually, the shell passes an arbitrary number of string arguments, where the number is only limited by the length of an input line. To make programming simple and uniform, the shell creates an array of pointers and only passes two values when it invokes a command: a count of command-line arguments and an array of pointers to character strings that constitute the arguments. Unix uses the names *argc* and *argv* for the two arguments a command receives from the shell; Xinu uses the names *nargs* and *args*. The names are only a convention — a programmer can choose arbitrary names for arguments when writing a function that implements a command.

The example shell adopts another convention from Unix: the first item in the *args* array is a pointer to the command name. An example will clarify the details. Consider the command line:

```
date -f illegal
```

Although the argument *illegal* is not permitted by the Xinu *date* command, the shell simply passes the string and allows the function that implements the *date* command to check its arguments. Figure 25.4 illustrates the two items the shell passes to the date function.

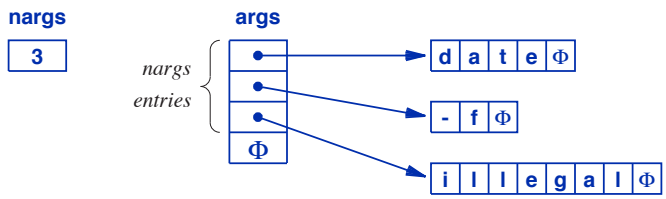



**Figure 25.4** Illustration of the two arguments (*nargs* and *args*) the shell passes to the date command for an input line: *date -f illegal*

Although passing an integer, such as *nargs*, to a command is trivial, the *args* array is more complex. In essence, the shell must construct the *args* array, and then pass its address to the command. There are two cases: builtin commands and non-builtin commands. We will consider builtins first.

After the shell has parsed the command line and removed tokens for I/O redirection and background processing, variable *ntok* will contain the count of remaining to-

kens, which is exactly the count needed for *nargs*. Furthermore, array *tok* contains the index in *tokbuf* where each token begins. Therefore, the shell can create an *args* array by computing the starting address of each token.

To form the *args* array, the code iterates through *ntok* tokens and for the $i^{th}$ token computes the expression:

```
&tokbuf[tok[i]]
```

That is, it sets *args[i]* equal to the address of the $i^{th}$ token in *tokbuf*. Once array *args* has been initialized, the shell calls the function that implements the builtin command.

## 25.14 Passing Arguments To A Non-Builtin Command

The second case, non-builtin commands, is more complex. Our shell creates a separate process to execute commands that are not builtin, and the command can execute in background (i.e., the shell can continue to read and handle new input lines while the command process runs in background). The question arises: what mechanism should the shell use to pass arguments to a process? The shell cannot use the same approach as with builtin commands because a command running in background needs a separate copy of its arguments that will not change as the shell goes on to process another command.

There are two ways to solve the problem of argument passing for a non-builtin command: the shell can allocate separate storage to arguments or the shell can hide the arguments in storage already allocated for the process. Because Xinu does not automatically release heap storage when a process terminates, the first approach requires the shell to keep a record of the argument storage allocated for each command so it can free the memory once the process completes. Thus, we have chosen the second approach:

> *After creating a process to execute a command, the shell places a copy of arguments in the stack area of the process and then allows the process to execute.*

Where on the process's stack should arguments be placed? Although it might be possible to rewrite *create* so it leaves space at the top of the stack, doing so is messy. Thus, we have chosen to use the area at the bottom of the stack. The shell stores a copy of the *args* array followed by a copy of the strings in *tokbuf* in the stack. Of course, pointers in the copy of the *args* vector must be assigned the addresses of strings in the copy of *tokbuf*. Figure 25.5 illustrates how the data from Figure 25.4 is arranged in contiguous memory locations.

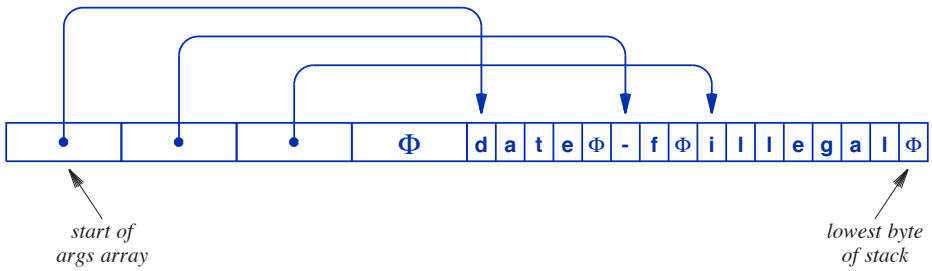



**Figure 25.5** A copy of the *args* array and argument strings in a process's stack.

Rather than incorporate code into the shell that copies items into the process's stack, our implementation uses a separate function, *addargs*. File *addargs.c* contains the code.

```
/* addargs.c - addargs */

#include <xinu.h>
#include "shprototypes.h"

/*------------------------------------------------------------------------
 *  addargs  -  add local copy of argv-style arguments to the stack of
 *                a command process that has been created by the shell
 *------------------------------------------------------------------------
 */
status  addargs(
          pid32         pid,            /* ID of process to use         */
          int32         ntok,           /* count of arguments           */
          int32         tok[],          /* index of tokens in tokbuf    */
          int32         tlen,           /* length of data in tokbuf     */
          char          *tokbuf,        /* array of null-term. tokens   */
          void          *dummy          /* dummy argument that was      */
                                        /*   used at creation and must  */
                                        /*   be replaced by a pointer   */
                                        /*   to an argument vector      */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        struct  procent *prptr;         /* ptr to process' table entry  */
        uint32  aloc;                   /* argument location in process */
                                        /*   stack as an integer        */
        uint32  *argloc;                /* location in process's stack  */
                                        /*   to place args vector       */
```

```
        char    *argstr;                /* location in process's stack  */
                                        /*   to place arg strings       */
        uint32  *search;                /* pointer that searches for    */
                                        /*   dummy argument on stack    */
        uint32  *aptr;                  /* walks through args array     */
        int32   i;                      /* index into tok array         */

        mask = disable();

        /* Check argument count and data length */

        if ( (ntok <= 0) || (tlen < 0) ) {
                restore(mask);
                return SYSERR;
        }

        prptr = &proctab[pid];

        /* Compute lowest location in the process stack where the       */
        /*      args array will be stored followed by the argument      */
        /*      strings                                                 */

        aloc = (uint32) (prptr->prstkbase
                - prptr->prstklen + sizeof(uint32));
        argloc = (uint32*) ((aloc + 3) & ~0x3); /* round multiple of 4  */

        /* Compute the first location beyond args array for the strings */

        argstr = (char *) (argloc + (ntok+1));  /* +1 for a null ptr    */

        /* Set each location in the args vector to be the address of    */
        /*      string area plus the offset of this argument            */

        for (aptr=argloc, i=0; i < ntok; i++) {
                *aptr++ = (uint32) (argstr + tok[i]);
        }

        /* Add a null pointer to the args array */

        *aptr++ = (uint32)NULL;

        /* Copy the argument strings from tokbuf into process's stack   */
        /*      just beyond the args vector                             */

        memcpy(aptr, tokbuf, tlen);
```

```
        /* Find the second argument in process's stack */

        for (search = (uint32 *)prptr->prstkptr;
             search < (uint32 *)prptr->prstkbase; search++) {

                /* If found, replace with the address of the args vector*/

                if (*search == (uint32)dummy) {
                        *search = (uint32)argloc;
                        restore(mask);
                        return OK;
                }
        }

        /* Argument value not found on the stack - report an error */

        restore(mask);
        return SYSERR;
}
```

Once a process has been created, the process table entry contains both the address of the top of the stack and the stack size. Because a stack grows downward in memory, *addargs* can compute the lowest memory address assigned to the stack by subtracting the stack size from the address of the stack top. However, a few details complicate the code. For example, because pointers must be aligned, *addargs* computes a starting location in the stack that is a multiple of four bytes. As a result, the final byte of the last argument string may end up to three bytes before the lowest byte of the stack. Furthermore, the code adds an extra null pointer to the end of the *args* array as shown in Figure 25.5.

Most of code in *addargs* operates as expected by computing the address in the stack at which the *args* array starts and then copying both the *args* array and the argument strings into the stack. However, the final *for* loop, which iterates through the process's stack may seem unusual: it finds the second argument that has been passed to the process and replaces it with a pointer to the *args* array. When the process is created, the shell uses a dummy value for the argument and then passes the value to *addargs* in parameter *dummy*. Thus, *addargs* searches the stack until it finds the value and replaces it.

Why does our implementation use a dummy argument and a search? The alternative consists of having *addargs* calculate the location of the second argument. Although calculating a location may seem cleaner, such a calculation requires *addargs* to understand the format of the initial process stack. Using a search means that only *create* needs to understand the details of process creation and the format of the stack. Of course, using a search has a disadvantage: the shell must choose a dummy argument

that will not occur early in the stack. Rather than choosing an arbitrary integer, our shell implementation uses the address of variable *tmparg*.

## 25.15 I/O Redirection

Once a process has been created to execute a command and the shell has called *addargs* to copy arguments into the process stack, all that remains is to handle I/O redirection and start the process executing. To redirect I/O, the shell assigns device descriptors to the array *prdesc* in the process table entry. The two key values are *prdesc[0]* and *prdesc[1]*, which the shell sets to *stdin* and *stdout*.

How do variables *stdin* and *stdout* receive values? Recall that the shell initializes them to *dev*, the device descriptor that was passed as an argument when the shell was invoked. Usually, the shell is invoked with device *CONSOLE*. Thus, if the user does not redirect I/O, the process executing a command will "inherit" the console device for input and output. If a user does redirect I/O, the shell sets variables *inname* and/or *outname* to the name that was specified on the command line. Otherwise *inname* and *outname* are set to NULL. Before assigning *stdin* and *stdout* to the command process, the shell checks *inname* and *outname*. If *inname* is non-null, the shell calls *open* to open *inname* for reading and sets *stdin* to the descriptor. Similarly, if *outname* is non-null, the shell calls *open* to open *outname* for writing, and sets *stdout* to the descriptor.

When should descriptors be closed? Our example code assumes that the command will close standard input and standard output descriptors before it exits; the shell does not clean up descriptors after the command completes. Forcing all commands to close their standard I/O devices before exiting has the disadvantage of making commands difficult to understand and difficult to program correctly because command code must remember to close devices even though the code does not open them. Having the shell monitor command processes and close standard I/O devices is also difficult because command processes are independent and multiple command processes can exit at the same time. The exercises suggest another alternative.

The final section of code in the shell runs the command process. There are two cases. To run the process in foreground, the shell calls *resume* to start the process, and then calls *receive* to wait for a message that the process has completed (when the process exits, *kill* sends a message to the shell). For the background case, the shell starts the command process but does not wait. Instead, the main shell loop continues and the shell reads the next command. The exercises suggest a modification of the code to improve correctness.

## 25.16 An Example Command Function (sleep)

To understand how a command processes arguments, consider function *xsh_sleep*, which implements the *sleep* command.† *Sleep* is trivial — it delays for the number of seconds specified by its argument. Thus, the delay is achieved by a single line of code

†By convention, the function that implements command *X* is named *xsh_X*.

that calls the *sleep* system function; the code is presented here merely to illustrate how arguments are parsed and how a command function prints a help message. File *xsh_sleep.c* contains the code.

```
/* xsh_sleep.c - xsh_sleep */

#include <xinu.h>
#include <stdio.h>
#include <string.h>

/*------------------------------------------------------------------------
 * xsh_sleep - shell command to delay for a specified number of seconds
 *------------------------------------------------------------------------
 */
shellcmd xsh_sleep(int nargs, char *args[])
{
        int32   delay;                  /* delay in seconds             */
        char    *chptr;                 /* walks through argument       */
        char    ch;                     /* next character of argument   */

        /* For argument '--help', emit help about the 'sleep' command   */

        if (nargs == 2 && strncmp(args[1], "--help", 7) == 0) {
                printf("Use: %s\n\n", args[0]);
                printf("Description:\n");
                printf("\tDelay for a specified number of seconds\n");
                printf("Options:\n");
                printf("\t--help\t display this help and exit\n");
                return 0;
        }

        /* Check for valid number of arguments */

        if (nargs > 2) {
                fprintf(stderr, "%s: too many arguments\n", args[0]);
                fprintf(stderr, "Try '%s --help' for more information\n",
                                args[0]);
                return 1;
        }

        if (nargs != 2) {
                fprintf(stderr, "%s: argument in error\n", args[0]);
                fprintf(stderr, "Try '%s --help' for more information\n",
                                args[0]);
                return 1;
```

```
        }

        chptr = args[1];
        ch = *chptr++;
        delay = 0;
        while (ch != NULLCH) {
                if ( (ch < '0') || (ch > '9') ) {
                        fprintf(stderr, "%s: nondigit in argument\n",
                                args[0]);
                        return 1;
                }
                delay = 10*delay + (ch - '0');
                ch = *chptr++;
        }
        sleep(delay);
        return 0;
}
```

## 25.17 Perspective

The design of a shell introduces many choices. A designer has almost complete freedom because a shell operates as an application that lies outside the rest of the system and only the command functions depend on the shell. Thus, as our example shows, the argument passing paradigm used by a shell can differ dramatically from the argument passing paradigm used throughout the rest of the system. Similarly, a designer can choose a syntax for command-line input as well as a semantic interpretation without affecting other parts of the system.

Perhaps the most interesting aspect of shell design arises from the choice of how much knowledge about commands is bound into the shell. On the one hand, if a shell knows all commands and their arguments, the shell can complete command names and check the command arguments, making the code that implements commands much simpler. On the other hand, allowing late binding means more flexibility because the shell does not need to change when new commands are created, but the tradeoff is that each command must check its arguments. Furthermore, a designer can choose whether to build each command function into the shell or to leave each command in a separate file, as Unix does.

Our example shell demonstrates one of the most important principles in shell design: a relatively small amount of code can provide powerful abstractions for a user. For example, consider how little code is needed to interpret input or output redirection, and the small amount of code needed to recognize an ampersand at the end of the line as a request to run a command in background. Despite their compact implementation, facilities for I/O redirection and background processing make a shell much more powerful and user-friendly than a shell in which each command interacts with a user to prompt for input and output information or whether to run in background.

## 25.18 Summary

We have examined a basic command-line interpreter called a *shell*. Although the example code is small, it supports concurrent command execution, redirection of input and output, and arbitrary string argument passing. The implementation is divided into two conceptual pieces: a lexical analyzer that reads a line of text and groups characters into tokens, and a shell function that checks the sequence of tokens and invokes a command.

The example code demonstrates the relationship between a user interface and the facilities provided by the underlying system. For example, although the underlying system provides support for concurrent processes, the shell makes concurrent execution available to a user. Similarly, although the underlying system provides the ability to open devices or files, the shell makes I/O redirection available to the user.

## EXERCISES

**25.1** Rewrite the shell to use *cbreak* mode and handle all keyboard input. Arrange for *Control-P* to move to the "previous" shell command, and interpret *Control-B* and *Control-F* as moving backward and forward through a line as Unix *ksh* or *bash* do.

**25.2** Rewrite the grammar in Figure 25.2 to remove the optional notation [ ].

**25.3** Modify the shell to allow I/O redirection on builtin commands. What changes are necessary?

**25.4** Devise a modified version of *create* that handles string arguments for the shell, automatically performing the same function as *addargs* when creating the process.

**25.5** Modify the shell so it can be used as a command. That is, allow the user to invoke command *shell* and start with a new shell process. Have control return to the original shell when the subshell exits. Be careful.

**25.6** Modify the shell so it can accept input from a text file (i.e., allow the user to build a file of commands and then start a shell interpreting them).

**25.7** Modify the shell to allow a user to redirect standard error as well as standard output.

**25.8** Read about shell variables in a UNIX shell, and implement a similar variable mechanism in the Xinu shell.

**25.9** Find out how the UNIX shell passes environment variables to command processes, and implement a similar mechanism in the Xinu shell.

**25.10** Implement inline input redirection, allowing the user to type

```
command << stop
```

followed by lines of input terminated by a line that begins with the sequence of characters *stop*. Have the shell save the input in a temporary file and execute the command with the temporary file as standard input.

**25.11** It would be possible to extend the command table to include information on the number and types of arguments each command requires, and to have the shell check arguments before passing them to the command. List two advantages and two disadvantages of having the shell check arguments.

**25.12** Suppose the designer decided to add a *for* statement to the shell so a user could execute a command repeatedly as in:

```
for 1 2 3 4 5 6 7 8 9; command-line
```

where *for* is a keyword, and *command-line* is a command line exactly like commands the shell now accepts. Should the designer modify the shell syntax and parser or try to make *for* a builtin command? Explain.

**25.13** Add command expansion to the shell by having the user type a unique prefix of a command followed by *ESC*; have the shell type out the completed command and wait for the user to add arguments and press the *Enter* key.

**25.14** UNIX allows command lines of the form:

```
command_1 | command_2 | command_3
```

where the symbol |, called a *pipe*, specifies that the standard output of one command is connected to the standard input of the next command. Implement a *pipe* device for Xinu, and modify the shell to allow a pipeline of commands.

**25.15** Modify the *tty* device driver and shell so that pressing CONTROL-c kills the currently executing process.

**25.16** Modify the *tty* device driver and shell so that pressing CONTROL-z places the currently executing process in background.

**25.17** Modify the design to permit I/O redirection and background processing for builtin commands: if redirection or background processing is needed, treat the command as a normal command and create a separate process.

**25.18** The text describes the problem of closing device descriptors when a command exits. Modify the system so *kill* automatically closes a process's descriptors when the process exits.

**25.19** The example shell calls *receive* to wait for a foreground process to exit, but does not check the message received. Show a sequence of events that can cause the shell to proceed before a foreground command has completed.

**25.20** Modify the code in the shell to repair the problem in the previous exercise.

# Appendix 1

# Porting An Operating System

*Progress, far from consisting in change, depends upon retentiveness....*

— George Santayana

## A1.1 Introduction

Previous chapters focus on the interior of an operating system. They present abstractions, discuss design tradeoffs, show how the code fits into a hierarchical organization, and examine implementation details. Chapter 24 examines how a system can be configured to allow the code to run on systems with a variety of peripheral devices.

This appendix examines two larger questions. First, how can an existing operating system be ported to a new machine or to a hardware platform that differs in a fundamental way? Second, can an operating system be written in a way that makes porting easier? To answer the first question, the appendix discusses cross-development and downloading, outlines the steps involved in understanding hardware, and provides practical advice about how to proceed. To answer the second question, the appendix discusses techniques that have been used to make an operating system adaptable.

## A1.2 Motivation: Evolving Hardware

Although work on operating systems demands the ability to grasp high-level abstractions, design efficient mechanisms, and understand small details, the most significant challenge facing operating system designers does not arise from the intellectual difficulty of the task. Instead, it arises from the constant changes in technology and the consequent economic pressure for vendors to create new products or to add features to existing products. In a fourteen-month period after we started a revision of this text, for example, a hardware vendor changed models twice. In one case, the change was dramatic — the processor chip, instruction set, memory organization, and I/O devices were completely replaced.

Because an operating system interfaces directly with the underlying hardware, even small changes in the hardware can have an overwhelming effect on the system. For example, if a vendor changes the hardware to reserve a piece of the memory address space for Flash ROM, the memory management software in the operating system must be modified. Although such modification may seem straightforward, the details may involve changes to page tables and the code that interacts with the MMU hardware as well as the code that allocates memory on demand. If a significant amount of the address space becomes reserved, the operating system's allocation policy may need to change. The point is:

> *Because both technological and economic factors cause continual changes in hardware, an operating system designer must be prepared to port systems to new platforms.*

## A1.3 Steps Taken When Porting An Operating System

Despite the effects of hardware change, moving an existing operating system to a new platform is much easier than designing and building a new system from scratch. In particular, if an operating system has been written in a high-level language, porting the system to a new platform is easy because a compiler can do most of the work.

Consider Xinu. Most of the code is written in C. If a C compiler is available for the new platform, many of the functions can be compiled without making changes in the source. If a given function deals with basic data structures, such as integers, characters, arrays, and structures, a compiler may be able to compile the code without change, and the resulting binary program may run correctly. Even in cases where change is needed, the modifications may be minor (e.g., to accommodate slight differences in compilers). Thus, a principle is:

> *An operating system written in a high-level language, such as C, is much easier to port to a new platform than a system written in assembly language.*

We will assume that, whenever possible, an operating system has been written in C, and we will consider the steps taken when porting to a new platform. Specifically, Figure A1.1 lists steps taken when porting Xinu.

| Step | Description |
|---|---|
| 1. | Learn about the new hardware |
| 2. | Build cross-development tools |
| 3. | Learn the compiler's calling conventions |
| 4. | Build a bootstrap mechanism |
| 5. | Devise a basic polled output function |
| 6. | Load and run a sequential program |
| 7. | Port and test the basic memory manager |
| 8. | Rewrite the context switch and process creation functions |
| 9. | Port and test the remaining process manager functions |
| 10. | Build an interrupt dispatcher |
| 11. | Port and test the real-time clock functions |
| 12. | Port and test a tty driver |
| 13. | Either port or create drivers for other devices |
| 14. | Once a disk is available, port a file system |
| 15. | Once a network driver is operational, port protocol software |
| 16. | Port a shell and other applications |

**Figure A1.1** The steps taken when porting Xinu to a new platform.

Note the relationship between the steps listed and the Xinu hierarchy. In essence, porting follows the same pattern as design: lower levels of the hierarchy are ported first, and successive levels are then added. The next sections highlight each step.

### A1.3.1 Learning About The Hardware And Compile

Step 1 may seem straightforward. Unfortunately, some vendors are reluctant to reveal details about their commercial hardware or software. Even if generic information is available (e.g., the processor instruction set), a vendor may choose to keep details secret (e.g., a map of the bus address space, the hardware initialization sequence, or the details about devices). Nevertheless, throughout the remainder of this appendix, we will assume that the needed information can be obtained.

### A1.3.2 Cross Development

If the target hardware platform already runs a fully-functional operating system, it may be possible to skip steps 1 through 6, and use the existing mechanisms to compile and boot a new system. In most cases, however, the target platform will be new, may lack the power needed for a production system, or may not be available for development. Thus, an operating system designer usually does not rely on the target platform to support software development. Instead, a designer uses a *cross-development* approach in which the compiler and linker are designed to produce code for the target machine, but the development tools run on a conventional computer.

One of the most widely used cross-development environments consists of the *Gnu C Compiler*, *gcc*. A copy of *gcc* can be downloaded and used at no cost from:

http://gcc.gnu.org

After downloading the source code for *gcc*, a programmer must select configuration options to specify details, such as the target processor type and the endianness of the target machine. The programmer runs the Unix utility *make* to build a version of the compiler, assembler, and linker that will produce code for the target machine.

### A1.3.3 Calling Conventions

Function invocation forms one of the most important aspects of porting. To build a context switch, for example, a programmer must precisely understand all details of the calling conventions. Although hardware designers include subroutine invocation mechanisms, understanding the hardware is not enough because a compiler can impose additional requirements.

It may seem that using an open source compiler means the calling conventions are obvious. However, an operating system designer needs to know about special cases, and the answers to questions may be difficult to find. Fortunately, information is often available on the web.

### A1.3.4 Bootstrap Mechanisms

Once it has been compiled and linked, a program image must be downloaded into the target platform. Early embedded hardware required that the image be burned into ROM, and ROM installed in a socket. Fortunately, modern systems use alternative mechanisms that require much less effort. Typically, the hardware includes bootstrap functionality that can read an image from a CD-ROM, accept an image over a console serial line, or download an image over a network. The bootstrap procedure is usually intended for system developers, and may not generally be known.

For example, consider the Linksys E2100L wireless router used as the platform for code in the text. Access to the boot loader is not available unless one opens the case, connects a serial line, and uses the serial line to send characters during the power-up sequence. Once the normal boot sequence has been interrupted, the boot loader presents a prompt and offers a variety of ways to download an image (e.g., multiple formats are available for downloading over the console serial line as well as network bootstraps available for downloading over an Ethernet interface). Whatever method is chosen, it is essential to find a way to place a copy of an image in memory on the target machine.

### A1.3.5 Polled Output

The next step in porting requires a programmer to devise a way for a running program to output characters. Until some basic I/O is available, a programmer must work in the dark — hoping that the image has been downloaded and started correctly. Basic I/O is invaluable: once even basic I/O is available, a programmer can determine how much of the system is working, and can isolate problems quickly.

Because early test programs do not include interrupt processing, the basic I/O mechanism must use *polling*. That is, a programmer creates a function similar to *kputc* that waits for an I/O device to become ready and then transmits a character. Both ends must agree on details such as the baud rate and bits per character, which can make debugging tedious. To simplify the code, the first version of *kputc* can be written in assembly language and can have information about the device (e.g., the CSR address and the baud rate) hardwired into the program.

### A1.3.6 Sequential Program Execution

Once an image can be downloaded and run, the next step consists of building an environment that permits a sequential program to execute. In particular, successful execution of a C program requires that memory permissions are set correctly (the program text is readable and data locations can be read and written) and a run-time stack exists (which is needed for function calls).

Initializing the environment may seem trivial, but it requires knowledge of many hardware details. For example, on the E2100L, physical memory is replicated in the ad-

dress space. To set a stack pointer to a high physical memory address, it is not sufficient merely to determine that a particular address is valid; one must also determine how each address corresponds to physical addresses.

### A1.3.7 Basic Memory Management

Once the layout of memory is known and a sequential program can be downloaded and run on the target hardware, a programmer can port and test the four basic memory management functions: *getmem*, *freemem*, *getstk*, and *freestk*. In addition to basic allocation and deallocation tests, a programmer should concentrate on alignment. Some hardware platforms require all memory accesses to be word aligned and some allow unaligned access. On machines that require alignment, a programmer should insure that the memory free list has been initialized in such a way that alignment works correctly (i.e., all allocated blocks begin on the appropriate boundary).

### A1.3.8 Process Creation And Context Switch

Once basic memory management is working, a programmer can begin to port the process manager functions. In particular, a programmer starts with context switch, scheduling, and process creation. A giant step forward occurs once the three fundamental process management functions are in place: instead of a sequential program, the code will be a fledgling operating system that supports concurrent execution.

There are two difficult parts in the design. Creating the saved information for a process requires an intricate knowledge of the machine state and the operation of the context switch. Building a context switch is tricky because it involves finding a way to save all the state associated with a process and reload all the state from another process. It can be easy to overlook details or inadvertently to destroy state while saving a copy (e.g., clobber a register). Unfortunately, debugging can be extremely difficult because problems may not be discovered until the system attempts to reload saved state.

### A1.3.9 Synchronization And Other Process Management Functions

Once process creation, scheduling, and context switching are working, other process management functions can be added easily. Semaphores functions can be ported and tested, as can message passing. Beyond the context switch level, most process management functions do not depend on the hardware. Of course, various data types may change, depending on the underlying hardware. For example, when moving from a 32-bit computer to a 64-bit computer, the *msg32* type may be changed to *msg64*. Nevertheless, porting the semaphore and message passing functions is a relatively straightforward task.

### A1.3.10 Interrupt Dispatching And Real-Time Clock

The last big hardware hurdle concerns interrupts. Building an interrupt dispatcher requires a detailed knowledge of the hardware. How do the processor, co-processor, and bus interact? Exactly what state does the hardware save when an interrupt occurs, and what state is the operating system required to save? How does the dispatcher determine which device interrupted? How does a dispatcher return to the running program when the interrupt ends? What addresses are used for the bus and devices?

The details of interrupts are surprisingly subtle. On many systems, for example, I/O is memory mapped. Thus, I/O devices (and perhaps the bus hardware) are mapped into specific addresses. To access I/O facilities, however, an operating system may need to disable or avoid the memory cache because each I/O access must go to the underlying hardware rather than the cache.

In any case, once interrupt dispatching is in place, an example is needed to test the mechanism. It is logical to start by testing the real-time clock. On some systems building a real-time clock handler first is necessary because the clock cannot be stopped — if the system has interrupts enabled, clock interrupts will occur. Clock interrupts mean that processes can call *sleep()* to delay for a specified time, and that time slicing is in effect.

### A1.3.11 Tty Device Driver

Clock interrupts are distinct from other devices because a clock does not perform output. A serial line is perhaps the simplest type of device that has both input and output (some hardware separates input and output interrupt handling). Thus, the tty driver will exercise both input and output, and insure that all basic interrupt processing works.

Fortunately, most systems include a serial line, and many use the same UART hardware as described in the text. Thus, much of the tty driver code, including the lower half, can simply be recompiled and used. Basic device parameters, which will have been worked out in Step 5, can be added to the device switch table or the lower half as appropriate.

### A1.3.12 Other I/O Software And Device Drivers

Once input and output have been tested, more complex device drivers can be ported. Devices that use DMA (e.g., disks and network interfaces) require buffer pools to be in place, and may require a deeper understanding of how DMA interacts with a memory cache. However, having a basic system in place makes debugging much easier because a programmer can focus on one device at a time.

### A1.3.13 File System

Given an operational disk driver, porting a basic file system is straightforward. The first step consists of porting and testing functions that read and write index blocks; the second step consists of porting and testing code that builds the free lists of index and data blocks. Once the basic allocation functions are in place, a directory can be added and the file system tested.

### A1.3.14 Protocol Software

A basic network device driver can be tested by sending and receiving raw packets (e.g., over an Ethernet). However, testing network connectivity and performance under load usually requires higher layers of protocol software, at least IP and UDP. With UDP in place, for example, a programmer can run an application on a conventional system that sends and receives a stream of packets.

### A1.3.15 Shell And Applications

Although applications are convenient, special programs are used to test each part of the operating system and exercise special cases. Once the system is running, the final step consists of porting a shell and more general-purpose applications.

## A1.4 Programming To Accommodate Change

How should operating system designers contend with constant change? Can we anticipate future hardware? Can a system be designed and implemented to make changes easier? Designers have been considering the questions for decades. Most early operating systems were created to match the hardware and written in assembly language. Each system was designed and built from scratch, with new abstractions and new mechanisms. As I/O devices (such as disks) and operating system abstractions (such as files) became standardized, designing a new system from scratch became much more expensive than adapting an existing system. Modern systems employ two techniques to accommodate change:

- Compile time: write source code that can generate multiple versions.
- Run-time: design facilities that allow an operating system to change dynamically.

*Compile time*. One way to make a system adaptable consists of writing source code that uses conditional compilation to allow a given source program to be used on multiple systems. As a simplistic example, consider writing an operating system that must run on hardware with a real-time clock or hardware that has no real-time clock. A programmer can use the C preprocessor to conditionally compile source code according

to the hardware. For example, if preprocessor variable *RT_CLOCK* has been defined, functions that use the clock should be compiled as usual. Otherwise, functions that depend on a clock should be replaced by versions that report an error. The *sleep* function from Chapter 13 illustrates the concept. To accommodate both situations, the code can be rewritten as follows:

```
syscall sleep(
          uint32        delay           /* time to delay in seconds     */
        )
{
#ifdef RT_CLOCK
        if (delay > MAXSECONDS) {
                return(SYSERR);
        }
        sleepms(1000*delay);
        return OK;
#else
        return SYSERR;
#endif
}
```

If constant *RT_CLOCK* has been defined, the C preprocessor generates the source code shown in Chapter 13, which is then compiled. If *RT_CLOCK* has not been defined, the C preprocessor eliminates the body of the *sleep* function and generates a single line of source code:

```
return SYSERR;
```

The chief advantage of conditional compilation lies in its efficiency: instead of using a test at run-time, the source code can be tailored to the specific hardware. Furthermore, the system does not contain extra code that is never used (which can be important in embedded systems).

*Run-time*. The simplest way to increase run-time portability consists of using conditional execution. When it starts, the operating system gathers information about the hardware and places the information in a global data structure. For example, a Boolean variable in the global data structure might specify whether the hardware includes a real-time clock. Each operating system function is written to interrogate the data structure and act accordingly. The chief advantage of using a run-time approach lies in generality — an image can be run without being recompiled.

The idea of run-time adaptation has been generalized by separating an operating system into two parts: a *microkernel* that contains basic process management functionality and a series of dynamically loaded kernel modules that extend the functionality. In theory, porting a microkernel to a new environment is easier than porting a complete

system because porting can be done piecemeal. That is, the microkernel is ported first, and modules are ported later, as needed.

## A1.5 Summary

Portability is important because hardware continues to change. The steps required to port an operating system to a new environment follow the same pattern as the original design: port the lower levels of the system first, and then port successively higher levels.

Operating system code can be written to increase portability. A compile-time approach that uses conditional compilation achieves highest efficiency. A run-time approach that uses conditional execution allows a single image to run on multiple versions of a platform. The most advanced run-time approach uses a microkernel plus dynamic kernel modules, which allows modules to be ported only if they are needed.

# Appendix 2

# Xinu Design Notes

## A2.1 Introduction

This appendix contains a set of informal notes that characterize Xinu and the underlying design. The notes are not intended as a tutorial, nor are they a complete description of the system. Instead, they provide a concise summary of characteristics and features.

## A2.2 Overview

*Embedded paradigm*. Because it is intended for use in embedded systems, Xinu follows a cross-development paradigm. A programmer uses a conventional computer (typically one running a Unix operating system, such as Linux) to write, edit, cross-compile and cross-link Xinu software. Output from cross-development software is an exact memory image. Once such an image has been created, a programmer downloads the image to the target system (typically over a computer network). Finally, the programmer starts the image running on the target embedded system.

*Source code organization*. Xinu software is organized into a handful of directories that follow the organization used with various Unix systems. Instead of placing all files for each module in a separate directory, files are grouped into a few directories. For example, all include files are placed in one directory and files that constitute the kernel sources are placed in another. Device drivers are the exception — source files for a given device driver are placed in a subdirectory named for the device type. The directories are organized as follows:

| | |
|---|---|
| compile | The Makefile with instructions needed to compile and link an image |
| include | All include files |
| config | Source for the configuration program and a Makefile that builds and installs *conf.h* and *conf.c* |
| system | Source code for Xinu kernel functions |
| devices | Source code for device drivers, organized into one directory for each device type |
| tty | Source code for the tty driver |
| rfs | Source code for the remote file access system, including the master remote file system device and remote file pseudo-devices |
| ether | Source code for the Ethernet driver |

## A2.3 Xinu Design Notes

*Xinu characteristics*. These are the notes kept during implementation; they are not intended to be a tutorial introduction to Xinu.

- The system supports multiple concurrent processes.
- Each process is known by its process ID.
- The process ID is used as an index into the process table.
- The system includes counting semaphores.
- Each semaphore is known by its ID, which is used as an index into the semaphore table.
- The system supports a real-time clock that is used for round-robin scheduling of equal-priority processes and timed delays.
- Each process is assigned a priority, which is used in scheduling; a process priority can be changed dynamically.
- The system supports multiple I/O devices and multiple types of I/O devices.
- The system includes a set of device-independent I/O primitives.
- The console device uses a tty abstraction in which characters are queued both during input and output.
- The tty driver supports modes; cooked mode includes character echo, erasing backspace, etc.

- The system includes an Ethernet driver that can send and receive Ethernet packets; the driver uses DMA.
- Xinu includes a local file system that supports concurrent growth of files without preallocation; the local file system only has a single-level directory structure.
- Xinu also includes a remote file system mechanism that allows access to files on a remote server.
- The system includes a message passing mechanism used for interprocess communication; each message is one word long.
- Processes are dynamic — a process can be created, suspend and restarted, or killed.
- Xinu includes a low-level memory manager used to allocate and free heap areas or process stacks, and a high-level memory manager used to create buffer pools, where each pool contains a set of fixed-size buffers.
- Xinu includes a configuration program that generates a Xinu system according to the specifications given; the configuration program allows one to choose a set of devices and set system parameters.

## A2.4 Xinu Implementation

*Functions and modules*. The system sources are organized as a set of functions. In general, each file corresponds to a system call (e.g., file *resume.c* contains system call *resume*). In addition to the system call function, a file may contain utility functions needed by that system call. Other files are listed below with a brief description:

| | |
|---|---|
| *Configuration* | A text file containing device information and constants that describe the system and the hardware. The *config* program takes file *Configuration* as input and produces *conf.c* and *conf.h*. |
| *conf.h* | Generated by *config*, it contains declarations and constants including defined names of I/O devices, such as *CONSOLE*. |
| *conf.c* | Generated by *config*, it contains initialization for the device switch table. |
| *kernel.h* | General symbolic constants and type declarations used throughout the kernel. |
| *prototypes.h* | Prototype declarations for all system functions. |
| *xinu.h* | A master include file that includes all header files in the correct order. Most Xinu functions only need to include *xinu.h*. |

| | |
|---|---|
| *process.h* | Process table entry structure declaration; state constants. |
| *semaphore.h* | Semaphore table entry structure declaration; semaphore constants. |
| *tty.h* | Tty line discipline control block, buffers, excluding sizes. |
| *bufpool.h* | Buffer pool constants and format. |
| *memory.h* | Constants and structures used by the low-level memory manager. |
| *ports.h* | Definitions by the high-level inter-process communication mechanism. |
| *sleep.h* | Definitions for real-time delay functions. |
| *queue.h* | Declarations and constants for the general-purpose process queue manipulation functions. |
| *resched.c* | The Xinu scheduler that selects the next process to run from the eligible set; *resched* calls the context switch. |
| *ctxsw.S* | The context switch that changes from one executing process to another. It consists of a small piece of assembler code that uses a trick: when the state of a process is saved, the execution address at which the process will restart is the instruction following the call to *ctxsw*. |
| *initialize.c* | General initialization and code for the null process (process 0). |
| *userret.c* | The function to which a user process returns if the process exits. *Userret* must never return. It must kill the process that executed it because the stack does not contain a legal frame or return address. |

## A2.5 Major Concepts And Implementation

*Process states*. Each process has a state given by field *prstate* in its process table entry. Constants that define process states have names of the form *PR_xxxx*. *PR_FREE* means the process entry is unused. *PR_READY* means the process is linked into the ready list and is eligible for the CPU. *PR_WAIT* means the process is waiting on a semaphore (given by prsem). *PR_SUSP* means the process is in hibernation; it is not on any list. *PR_SLEEP* means the process is in the queue of sleeping processes and will awaken

after a timeout. *PR_CURR* means that the process is (the only one) currently running. The currently running process is not on the ready list. *PR_RECV* means the process is blocked waiting to receive a message; *PR_RECTIM* is like *PR_RECV* except the process is also sleeping for a specified time and will awaken if the timer expires or a message arrives, whichever happens first.

*Counting semaphores*. Semaphores reside in the array *semtab*. Each entry in the array corresponds to a semaphore and has a count (*scount*) and state (*sstate*). The state is *S_FREE* if the semaphore slot is unassigned, and *S_USED* if the semaphore is in use. If the count is negative *P* then the head and tail fields of the entry in the semaphore table point to the head and tail of a FIFO queue of *P* processes waiting for the semaphore. If the count is nonnegative *P* then no processes are waiting and the queue is empty.

*Blocked processes*. A process that is blocked for any reason is not eligible to use the CPU. Any action that blocks the current processes forces it to relinquish the CPU and allow another process to execute. A process that is blocked on a semaphore is on the queue for the semaphore, and a process blocked for a timed delay is on the queue of sleeping processes. Other blocked processes are not on a queue. Function *ready* moves a blocked process to the ready list and makes the process eligible to use the CPU.

*Sleeping processes*. A process calls *sleep* to delay for a specified time. The process is added to a delta list of sleeping processes. A process may only put itself to sleep.

*Process queues and ordered lists*. There is a single data structure used for all process lists. The structure contains entries for the head and tail of each list as well as an entry for each process. The first *NPROC* entries in the table (0 to *NPROC-1*) correspond to the *NPROC* processes in the system; successive entries in the table are allocated in pairs, where each pair forms the head and tail of a list.

The advantage of keeping all heads and tails in the same data structure is that enqueuing, dequeuing, testing for empty/nonempty, and removing from the middle (e.g., when a process is killed) are all handled by a small set of functions (files *queue.c* and *queue.h*). An empty queue has the head and tail pointing to each other. Testing whether a list is empty is trivial. Lists can be ordered or may be FIFO; each entry has a key that is ignored if the list is FIFO.

*Null process*. Process 0 is a null process that is always available to run or is running. Care must be taken so that process 0 never executes code that could cause it to block (e.g., it cannot wait for a semaphore). Because the null process may be running during interrupts, interrupt code may never wait for a semaphore. When the system starts, the initialization code creates a process to execute *main* and then becomes the null process (i.e., executes an infinite loop). Because its priority is lower than that of any other process, the null process loop executes only when no other process is ready.

# NOTES

# NOTES

Computer Science & Engineering

**Operating System Design: The Xinu Approach, Linksys Version** provides a comprehensive introduction to Operating System Design, using Xinu, a small, elegant operating system that serves as an example and a pattern for system design. The book focuses the discussion of operating systems on the microkernel operating system facilities used in embedded systems. Rather than introduce a new course to teach the important topics of embedded systems programming, this textbook takes the approach of integrating more embedded processing into existing operating systems courses. Designed for advanced undergraduate or graduate courses, the book prepares students for the increased demand for operating system expertise in industry.

**Highlights**

- Explains how each operating system abstraction can be built and shows how the abstractions can be organized into an elegant, efficient design
- Considers each level of the system individually, beginning with the raw hardware and ending with a working operating system
- Covers every part of the system, so a reader will see how an entire system fits together, not merely how one or two parts interact
- Provides source code for all pieces described in the text, leaving no mystery about any part of the implementation — a reader can obtain a copy of the system to examine, modify, instrument, measure, extend, or transport to another architecture
- Demonstrates how each piece of an operating system fits into the design, in order to prepare the reader to understand alternative design choices

Beginning with the underlying machine and proceeding step by step through the design and implementation of an actual system, **Operating System Design: The Xinu Approach, Linksys Version** guides readers through the construction of a traditional process-based operating system using practical, straightforward primitives. It reviews the major system components and imposes a hierarchical design paradigm that organizes the components in an orderly and understandable manner.

*Software and instructions for building a lab that allows students to experiment are available on the author's website: www.xinu.cs.purdue.edu*

K13816

ISBN: 978-1-4398-8109-5

90000

9 781439 881095

CRC Press
Taylor & Francis Group
an informa business
www.crcpress.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
711 Third Avenue
New York, NY 10017
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK