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 peripherals such as display screens, disks, computer networks, and printers. Collectively, the code that performs control and coordination chores has been referred to as an *executive*, a *monitor*, a *task manager*, and 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 processor 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 executing intricately detailed, low-level hardware instructions. 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 the services offered and underlying hardware: an operating system provides impressively 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 the serial I/O device used

for a keyboard or mouse. 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:

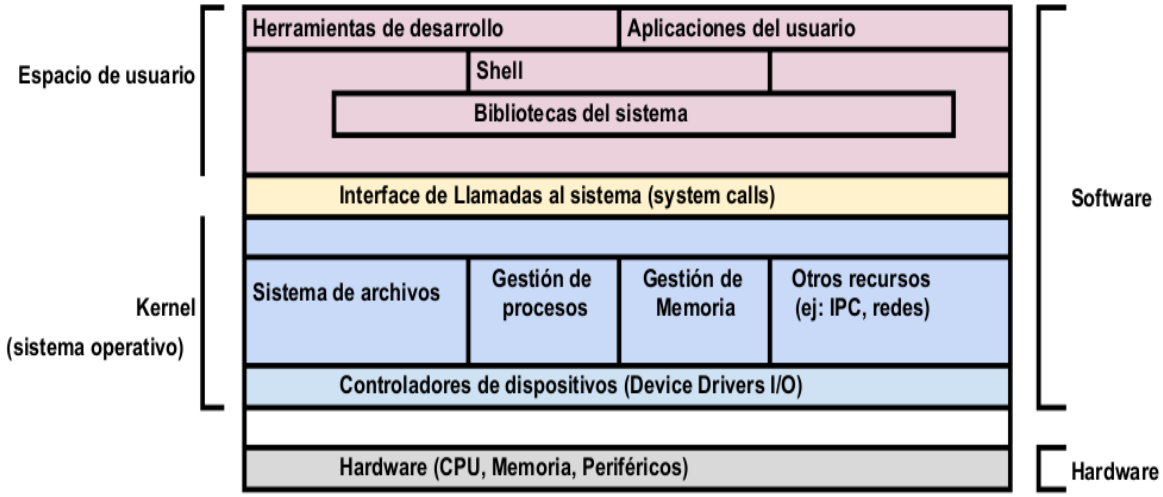
> *An operating system is designed to hide low-level hardware details and to create an abstract machine that provides applications with high-level services.*

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. By the time advances in micro-electronic technology reduced fabrication costs and made personal computers available, operating systems had become commodities, and few programmers need to 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 microprocessors 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 we use 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.



1.3 A Hierarchical Design

If designed well, the interior of an operating system can be as elegant and clean as the best application 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 runtime 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.

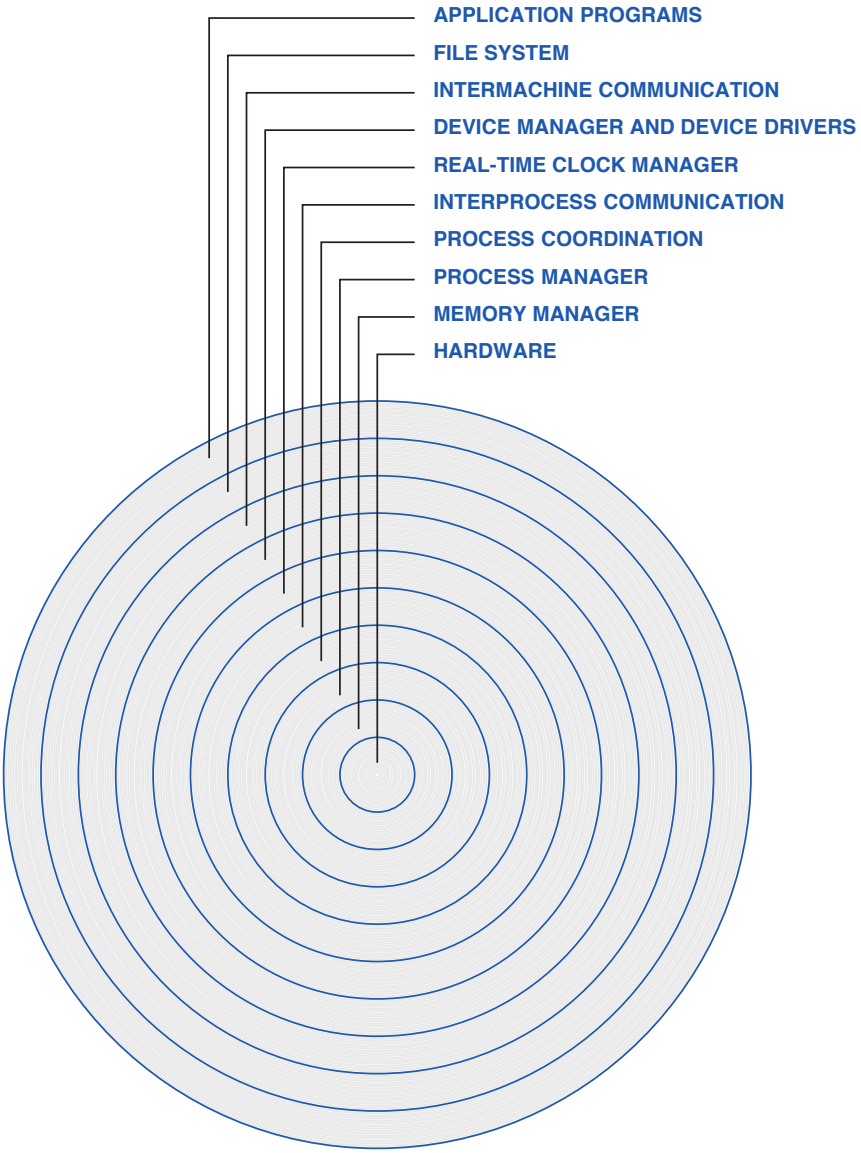


Figure 1.1 The multi-level organization used in the text.

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. Application programs occupy the highest conceptual level of the hierarchy — an application has access to all the facilities provided by lower levels.

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 runtime. 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 other 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 its printers until 2005. In each case, when the device was powered on, the hardware loaded a memory image that contained Xinu.

†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 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.

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. Almost all application programs use library functions, and the software found in libraries can offer significant convenience and functionality. Some operating systems even employ an optimization that allows code from a library to be loaded in memory and shared among all

applications. Despite the close relationship, library software remains independent of the underlying operating system.

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*). We will learn more about bootstrapping in Chapter 22.

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. At runtime, however, a system call and a conventional function call differ. Instead of transferring control to another function, 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 known as 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 function *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, '\n');
}
```

†The interface is also known as a *system call interface* or the *kernel interface*.

The code introduces several conventions used throughout Xinu. The statement:

```
#include <xinu.h>
```

inserts a set of declarations into 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* contains a series of *#include* statements that reference files needed by the Xinu system, 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 three characters to the console serial device: "h", "i", and a line feed (known as *NEWLINE*). The line feed is a control character that moves 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. A control character has 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 each function (*main*). Having a block comment before each function makes it easy to locate functions in a given file.

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, understanding how an operating system works internally helps a programmer appreciate concurrent processing and make sensible choices about system services. In some cases, the behavior of application software can only be understood, and problems can only be solved, by understanding how an operating system manages concurrent process execution.

The key to learning operating systems lies in persistence. A concurrent paradigm requires you to think about computer programs in new ways. Until you grasp the basics, it may seem confusing. Fortunately, you will not be overwhelmed with code — some of the most important ideas are contained neatly in a few lines of code. Once you understand what's going on, you will be able to read operating systems code easily, understand why process coordination is needed, and see how functions work together. By the end of the text, you will be able to write or modify operating systems functions.

1.9 Summary

An operating system provides a set of convenient, high-level services over low-level hardware. Because most applications use operating system services, programmers need to understand operating system principles. Programmers who work on embedded devices need to understand operating system design more deeply. Using a hierarchical structure can make an operating system easier to design, understand, and modify.

The text takes a practical approach. Instead of merely describing commercial systems or listing operating system features, it uses an example system, Xinu, to illustrate how a system can be designed. 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 function in the file system code use a function in the process manager? Can a function in the process manager use a function in the file system? 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 count the lines of source code used for the kernel.