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 small piece added to an operating system must fit the overall design. Therefore, the mechanics of assembling pieces forms only a minor 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 reinforce 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 research 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, wireless sensors, 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 multi-level hierarchy, 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 given 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 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 Xinu code presented in the text runs on many hardware platforms. We will focus on two low-cost experimenter boards that use two popular processor architectures: a *Galileo* board that contains an Intel (x86) processor and a BeagleBone Black that contains an ARM processor. The paradigm is that a programmer uses conventional tools (editor, compiler, and linker) to create a Xinu image. The image is then loaded onto a target board, and the board boots the Xinu operating system.

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, covering most of the material in one semester demands an extremely rapid pace usually unattainable by undergraduates. Few undergraduates are adept at reading code, 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. Choosing items to omit depends largely on the background of students who take the course. 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). If students have taken a data structures course that covers memory management and list manipulation, Chapters 4 and 9 can be skipped. It is important for students to understand that most operating systems include network communication. If they will take a separate course in networking, however, they can skip Chapter 17 on network protocols. 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, trade-offs, 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. Two topics should be included in both graduate and undergraduate courses: the important metamorphosis that occurs during startup when a sequential 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. The low cost of the boards we have selected (they are available for less than $50 US) means each student can afford to purchase a board and the cables needed to connect it to a laptop or other development computer. 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.) 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 paging 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. Of course, a background in programming is assumed — working on the code requires a knowledge of the C programming language and a basic understanding of data structures, including linked lists, stacks, and queues.

At Purdue, we have a lab with an automated system providing access to the experimenter boards. A student uses cross-development tools on a conventional Linux system to create a Xinu image. The student then runs an application that uses the lab network to allocate one of the boards, load the image onto the board, connect the console line from the board to a window on the student's screen, and boot the image. Because the hardware is inexpensive, a lab can be constructed at very low cost. For details, contact the author or look on the website:

www.xinu.cs.purdue.edu

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. Many basic ideas and names have been taken from Unix. However, 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. Xinu is not Unix.

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, and many students at Purdue contributed. As we updated the code, we strove to preserve the elegance of the original design. Rajas Karandikar and Jim Lembke created drivers and the multi-step downloading system used on the Galileo. Students in my operating systems class, including Andres Bravo, Gregory Essertel, Michael Phay, Sang Rhee, and Checed Rodgers, found problems and contributed to the code. Special thanks go to my wife and partner, Christine, whose careful editing and suggestions made many improvements throughout.

Douglas Comer