OPERATING SYSTEM DESIGN
VOL. I: THE XINU APPROACH

(PC EDITION)

DOUGLAS COMER

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

TIMOTHY V. FOSSUM

Department of Applied Computer Science
University of Wisconsin-Parkside
Kenosha, WI 53141

PRENTICE-HALL, INC.
Englewood Cliffs, New Jersey 07632

(Actual copyright and permissions page to be set by Prentice-Hall - don't forget to include the following statements).

To our families

Contents

Preface To The PC Edition

A singular development has led to this new edition of *Operating System Design - The Xinu Approach*. That development is the almost universal availability of IBM Personal Computers and compatible machines from other vendors, which we simply call PCs. These versatile and inexpensive machines are comparable in performance and memory capacity to the LSI-11 systems described in the first edition of this text. Moreover, PCs support a variety of programming tools sufficient to carry out the entire design and development of an operating system like Xinu on the PC itself. The PC makes it possible for small groups and individuals to study and experiment with operating systems without spending inordinate amounts of money on laboratory facilities.

Experience in teaching operating systems has convinced us that students learn best when they participate in the design, implementation, and modification of a real operating system. Hands-on laboratory work allows students to observe problems and solutions in detail. It gives them intuition and demonstrates the importance of abstractions. In this regard, Xinu works well as a laboratory tool. It is sufficiently sophisticated for use in working systems (e.g., volume 2 shows how it has been used to implement network communication and a user interface), yet it is simple enough for one person to understand in its entirety. The complete source code is available for a nominal charge and it compiles with either of the popular Microsoft or Turbo C compilers (see the tear-out card in the back of the book for information on ordering).

Like the previous edition, this book guides the reader through the design of a complete operating system, but uses the PC version of Xinu as its example. Although PC-Xinu has over 150 procedures and 8000 lines of code, it retains the remarkable simplicity that results from a hierarchical design. The text discusses design and alternatives, while the exercises suggest possible modifications. We encourage the reader to study the code carefully and to try as many of the exercises as possible.

One measure of an operating system design is how easily it adapts to new hardware architectures. Xinu has done quite well in making the transition across a variety of machines. It now runs on many Digital Equipment Corporation LSI 11, PDP 11, and VAX machines; Apple Computer Corporation MacIntosh, MAC plus, and MAC 2; Sun Microsystems Inc. Sun 2 and Sun 3; National Semiconductor 32000-based machines; and, as this text shows, on the PC. Almost the complete design, as well as much of the code, remains unchanged across all implementations, demonstrating its flexibility.

While the PC presents an opportunity, it also poses an obstacle. The hardware and software configuration of the PC is not well suited to a multi-tasking environment and, in many ways, makes concurrent execution more difficult than necessary. We have elected

to use the PC Basic Input/Output System (BIOS) that vendors usually supply in read-only memory (ROM) to implement low-level I/O operations. Unfortunately, the procedures supplied by the ROM BIOS are not reentrant, and avoiding reentrancy problems results in awkward interrupt handlers. Using ROM BIOS calls also makes the operating system execute slower than a system that handles hardware-level input/output directly. However, using the ROM BIOS has an overwhelming quality: it avoids incompatibilities among vendor's hardware, allowing the same operating system to run on a wide variety of PC brands.

We have omitted the Xinu ring network from this edition for two reasons. First, newer technologies make it obsolete. Second, volume 2 provides a more complete discussion of networks and the integration of protocol software into an operating system. Two new chapters replace the two network chapters. One discusses windows and the other discusses an interface to the MS-DOS file system. Readers familiar with the PC will find both informative because they show how easy it is to accommodate such features in the operating system design.

Chapters 1-13 of this edition follow the topics of the first edition, with changes limited to the code. For example, the 8088 processor uses a 16-bit FLAGS register, while the LSI-11 stores processor status in the low-order 8 bits of its program status word. To accommodate the difference, we have introduced minor changes in the disable and restore procedures that save and restore processor status. In making changes, our goal has been to preserve the structure of the system wherever possible. For example, because the PC uses a memory-mapped video display, console output cannot be interrupt driven as in most versions of Xinu. We elected to retain the Xinu device and buffer structure by substituting a process that moves characters to the display memory in place of the lower-half interrupt routines.

Chapter 14, new to this edition, develops a simple window mechanism that permits non-overlapping rectangular screen areas on the PC's video display to serve as logical tty devices. Although not as sophisticated as the bit-mapped graphics available on some computers, our window scheme demonstrates the basic principles.

Later chapters follow the LSI-11 edition. Chapters 15 through 17 cover higher-level memory management and queued messages, a low-level disk driver, and a Xinu file system. A new Chapter 18 develops routines that provide access to MS-DOS files, putting them in the same conceptual and operational framework as Xinu files.

In addition to those who helped with the first edition, we gratefully acknowledge the contributions of many people whose assistance made this edition possible. At Purdue, so many people have worked on Xinu over the past six years that their contributions cannot be enumerated individually. Andy Thomas initially transported Xinu to the IBM PC and made his work available to others. Charlotte Tubis edited the manuscript extensively and suggested ways to improve wording.

At the University of Wisconsin-Parkside, undergraduate students helped refine the details of the PC-Xinu implementation. Tim Knautz helped design and implement window devices and found several bugs in an early version of PC-Xinu.

Bob Duchesneau devised a pipe mechanism similar to UNIX pipes, Erhard Trudrung implemented receive with timeout, Andy Krieg added RAM disk support, and Dave Datta built a printer device driver.

Dave Brown, who also configured in multiple disk devices, file protection modes and file time-stamps, hooked the pieces together, tested it mercilessly, and found additional bugs.

Marty Wegner uncovered a number of inconsistencies and conceptual lapses that resulted in the most recent improvements to the code. He also implemented extensive enhancements, including the Volume II shell running concurrently in multiple windows.

Paul Sorensen and Gene Lai kept the machines in the UW-Parkside Operating Systems Laboratory going in spite of the misuse to which they were subjected.

Chris Comer patiently and carefully read several drafts for errors and made many good suggestions. Ella Fossum contributed more patience and understanding than could reasonably be expected from the wife of a co-author.

Zenith Data Systems kindly provided Z-158 PC and Z-241 AT-type machines used to develop PC-Xinu.

Finally, we are indebted to Purdue University and the University of Wisconsin-Parkside for their support.

Douglas Comer and
Timothy Fossum

January 26, 1988

Foreword To The First Edition

The quasars are so remote and the quarks so minute, it seems impossible to comprehend all dimensions of our physical universe. For example, the most distant quasar is on the order of 10^{27} meters distant and the quarks are on the order of 10^{-18} in diameter. This is a span of 45 orders of magnitude from the largest known distance to the smallest. The ratio of the largest to the smallest known distance, 10^{45}, is miniscule when compared to the number of subatomic particles in the universe, approximately 10^{80}.

When physical constraints are removed, human thought will routinely attempt to grapple with numbers incomprehensibly larger than these. For example, the largest known Fibonacci number exceeds 10^{208} and the largest known prime exceeds 10^{3374}. One of the largest numbers to appear in a mathematical proof (Skewes' number) is on the order

$$10^{10^{10^{34}}}$$

The combinatorial spaces through which our computer programs must search for solutions can easily reach such staggering sizes.

How can the human mind overcome barriers of such incomprehensible size? How can we assemble comprehensible computers and programs capable of dealing with vast spaces of abstract objects, the largest of which are many orders of magnitude bigger than the smallest?

Nature long ago solved this problem. It is the hierarchical ordering principle of building up structures by levels and clusters. Issac Asimov has, over the years, explored this principle by studying ''ladders'' measuring the universe by length, area, volume, mass, density, pressure, time, speed, and temperature.† There is a film, *Powers of Ten*, that explores the ladder of length with fascinating visual simulations; it can be seen in a display at the Smithsonian Air and Space Museum in Washington, DC.

The *Powers of Ten* film simulates a viewscreen spanning a distance of 10^n meters for a range of integer ''steps'' on a ladder of length n. Initially, the viewscreen is 1 meter wide (Step 0) and shows a picnicker on a Chicago football field. The viewer ascends the ladder of length as the viewscreen expands its scope by a factor of 10 every 10 seconds. For example, at Step 2, the screen contains the entire football field; at Step 5, the Chicago metropolitan area; at Step 8, the planet earth; at Step 13, the solar system; at Step 19, the local star group; at Step 21, the Milky Way Galaxy; and at Step 22, the local cluster of galaxies. Finally, at Step 27, the screen shows a few faint blips, each a quasar or cluster

†See *The Measure of the Universe*, Harper and Row, New York, 1983; and *Asimov on Numbers*, Pocket Books, New York, 1977.

of galaxies. The film then returns to the starting point and begins descending the ladder by shrinking the screen's scope by a factor of 10 every 10 seconds. For example, at Step -2, the screen shows a small patch on the picnicker's arm; at Step -5, individual cells in the picnicker's skin; at Step -9, molecules; at Step -10, atoms; at Step -13, protons and neutrons. Recent advances in physics have measured quarks, the smallest subatomic particles known, at Step -18.

These explorations of the universe, ascents and descents on ladders of measure, show the same striking patterns:

a. The universe is mostly empty space.

b. At each step there are well-defined objects with well-defined rules of interaction.

c. The objects of a given step are composed of objects of lower steps and are constituents of objects of higher steps.

The hierarchical ordering principle – *the Principle of Steps* – is nature's way of structuring the universe. It applies across all known steps of all ladders of measure, and it probably applies at new steps yet to be discovered.

By the middle 1960s, computer scientists were becoming seriously concerned about the sizes of computer programs. On a ladder of storage, the largest user programs of the day were Step 5 – i.e., when compiled, occupied on the order of 10^5 words of file store. The largest programs of all, operating systems, were then at Step 6 and were threatening to reach Step 7. You will not be surprised, then, when I tell you that, of all computer scientists, operating systems designers were the most anxious to apply nature's Principle of Steps to impose order on large programs.

At the First ACM Symposium on Operating Systems Principles (SOSP) in 1967, Dijkstra reported organizing the software of the THE operating system into seven levels (steps). Each level consisted of a collection of abstract objects and a set of rules governing their behavior. (The rules were enforced by encoding them into operating system procedures called ''primitive operations'' or simply ''primitives.'') The internal structure of an object at a given level was invisible at that level but could be explained in terms of objects and operations of levels below. Dijkstra had applied nature's Principle of Steps to the design of a large program and, in so doing, had constructed a comprehensible Step 5 operating system.

The Principle of Steps was employed in other experimental operating systems such as Liskov's VENUS operating system (4 levels, 1972) and SRI's Provably Secure Operating System (17 levels, 1975). This principle has influenced programming language design as well, under the somewhat mystical name of ''data abstraction;'' examples include Simula (1967), Concurrent Pascal (1975), Ada (1980), and Smalltalk (1981). It has influenced software engineering, under the name ''top-down refinement.'' It has influenced computer communications, under the name ''layered network protocols.''

Despite its presence in some rivulets of computer science for many years, the Principle of Steps has been slow to flow into the mainstream of computer systems design. Skeptics argue that systems constrained by this principle are inherently less efficient than systems constructed by expert programmers not so constrained. They offer two lines of argument. The first, which shows up most clearly in ‘‘layered network protocols,’’ is that each level of software has access *exclusively* to the next lower level. This leads to a design in which messages must be passed down through intermediate levels to gain access to objects several steps down in the hierarchy. In fact, the Principle of Steps merely requires objects to be composed solely of lower objects; it does not rule out direct access to any visible lower object. The second argument is that constraints on structure increase system size by removing opportunities for optimizations. This remarkably persuasive argument has never been supported by data. In fact, all operating systems designed with explicit attention to hierarchical ordering have been significantly *smaller* than other systems of similar function.

Doug Comer’s book, – this book – is about XINU, an operating system for a set of LSI 11 computers capable of cooperative computation via a store-and-forward ring network. (XINU stands for ‘‘XINU is not UNIX,’’ and is pronounced ‘‘zee-new.’’) XINU incorporates the concepts of UNIX into a level-structured operating system that can run in as little as 4000 bytes (2000 words) of main store. The entire set of XINU source files include, with comments, just under 5850 lines of C code and 650 lines of assembler code. (Without comments, there are 4300 lines of C code and 550 lines of assembler.)

So there you have it: a real, Step 5 operating system with all the functionality of the Step 7 operating systems of the early 1970s. Pretty impressive, isn’t it? Nature’s Principle of Steps works.

Peter J. Denning

August 7, 1983

Preface To The First Edition

Building a computer operating system is like weaving a fine tapestry – it consists of producing a large, complex object in many small steps. Like stitches in a tapestry, details are important because mistakes are noticeable. But understanding details and the mechanics of assembling pieces is only a small part of the problem; a masterful creation requires a pattern that the artisan can follow.

Surprisingly, few operating system textbooks or courses explain that there is a pattern from which systems can be built. Some students still hear the rhetoric that often was taught a decade ago: ‘‘operating system design is mostly black art and little science.’’ Textbooks reenforce these ideas by focusing on details that have especially elegant explanations, independent of how such topics pertain to modern systems. As a result, students are left with the feeling that operating systems consist of a few well-understood pieces that are somehow connected by what is otherwise a morass of mysterious code containing many machine-dependent tricks.

Now that inexpensive microprocessors have become abundant more programmers are being asked to design software systems starting with the bare machine. It is important that programmers working with such hardware know the fundamentals of operating system design for two reasons. First, operating system primitives provide incredible intellectual leverage – it is impossible to devise systems that exploit the power of these new computers without understanding operating systems. Second, the effort that has been expended in operating system research is staggering – it is unlikely that any programmer would ever stumble onto a good design without rigorous training.

This book attempts to remove the magic from operating system design and to consolidate the body of material into a systematic discipline. It reviews the major system components and a structure that organizes them in an orderly, understandable manner. Unlike texts that survey the field by presenting as many alternatives as possible, it guides the reader through the construction of a conventional process-based system, using practical, straightforward primitives. It 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 of and a pattern for system design.

Although Xinu is small enough to fit into the text, it 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 drivers, intermachine communication (networks), and a file system. These 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. Readers accustomed to commercial microcomputer ‘‘operating systems’’ will be pleasantly surprised by its sophistication. An important lesson to be learned is that good system design can be as important on small machines as it is on large ones.

With only a few exceptions, the book covers topics in the sequence that a designer follows when building a system. Each chapter describes a component in the design hierarchy and presents software that illustrates how to implement primitives in that layer. This approach has several advantages. First, each chapter explains a larger subset of Xinu than the previous ones, making it possible to think about the design and implementation of a given layer independent of the implementation of preceding or succeeding layers. Second, the details of any chapter can be skipped on first reading – a reader need only understand what services the routines in that chapter (layer) provide, not how those routines are implemented. Third, the reader sees the implementation of a procedure before that procedure is used to build others, making clear how each layer is built out of previous ones. Fourth, intellectually deep subjects like concurrency come up early, before many procedures have been introduced, while the bulk of the code (intermachine communication and file systems) comes at the end when the reader is better prepared to understand the details.

Chapters 1-13 describe a ‘‘minimal’’ system that supports concurrent processing, terminal input and output, and real-time clock management. Although the minimal system may not seem useful at first, it has served as the basis for several applications, including a VLSI chip tester. Later chapters describe machine-to-machine communication (computer network) software and the file system that are built on top of the minimum system.

Unlike many other books on operating systems, this one 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 should be easy to master.

The book is designed for advanced undergraduate or graduate-level courses. Although there is nothing inherently difficult about any topic, covering most of the material in one semester demands a rapid pace (usually unattained by undergraduates).

In lower-division system courses, class time may be needed to help students understand the motivation and details that are presented. Although such exposition may seem unnecessary, experience has shown that students at this level find concurrency an extremely difficult notion. Many are not adept at reading sequential programs, and fewer still really understand the details of a run-time environment or machine architecture; they need to be guided through the chapters on process management carefully. It helps immensely if students have hands-on experience with the system so they can observe it in action. The host software runs on a VAX computer under the UNIX operating system.

Ideally, students will have the opportunity to *use* Xinu during the first few day or weeks before they try to understand its internal structure. Chapter 1 provides a few examples and encourages experimentation. (It is surprising how many students take system courses without ever writing concurrent programs.)

In advanced courses, students understand concurrent programming and machine architecture. They can pick up details from the text, leaving time in the classroom to discuss alternative sets of primitives, alternative implementations, and proof of correctness. Students should be encouraged to read some of the many journal articles and books on operating systems, and to see how the primitives in Xinu extend into more complex hardware systems.

Programming projects are strongly encouraged at all levels. Many exercises suggest modifying or measuring the code, or trying alternatives. (The software is available for a nominal charge). Many of the exercises suggest improvements, experiments, and alternative implementations. Larger projects are also possible. Examples that have been used include: a virtual circuit protocol layer to go on top of the present datagram layer; the design of an internet naming and addressing scheme; a remote file server; a remote login facility to allow machines to log into a host operating system across the ring network; and the design of a text editor that minimizes the cost of sending files across the network. Other students have transported Xinu to processors like the Intel 8086 and Motorola 68000.

Some background in basic programming is assumed. The reader should understand basic data structures like linked lists, stacks, and queues, and have written programs in a high-level language like Pascal, PL/I, or C.

I encourage designers to code in high level languages whenever possible, reverting to assembly language only when necessary. Following this approach, I have written most of Xinu in C. Appendix 1 contains a quick introduction to C for readers who are interested only in reading the programs. It explains C constructs by comparing them to similar constructs in Pascal. Readers have an opportunity to develop their ability to read C code in Chapter 3 which deals with a familiar subject (linked lists). The linked list procedures form an especially easy introduction to C because they do not contain any explicit references to concurrent process control constructs. Readers who want to write programs or make substantial changes to Xinu can find more detail about C in the book by Kernighan and Ritchie [1978].

A few machine dependent routines are written in LSI 11 assembler language (almost identical to that of the popular PDP 11). However, the explanations and comments accompanying these routines make it possible to understand them without learning assembler language in detail.

I gratefully acknowledge the help of many people who contributed ideas, hard work, and enthusiasm to the Xinu project. At Purdue, a group of graduate students helped with the initial design and implementation. They also gathered together most of the cross-development software.

Andre Bondi and Subhash Agrawal read through early versions of the process manager and context switch code before a viable compiler and downloader were available.

Steve Salisbury built a C library for Xinu that was compatible with the UNIX C library.

Matt Bishop, Ken Dickey, and Bhasker Parathasarathy adapted a PDP 11 C compiler to the LSI 11.

Dave Schrader devised a process structure for the store-and-forward ring network, and suggested the ports mechanism for passing frames between layers.

Sean Arthur and Vincent Shen spent many hours wiring together a reconfigurable star-shaped ring network and connections to the host computer.

Derrick Burns, a student at Princeton University, transported Xinu to a Motorola 68000 system.

Bob Brown and Chris Kent wrote the downloader, uploader, and post-mortem debugger. Both made helpful suggestions about the choice of primitives and the implementation details. Bob put together an early version of *kprintf* that helped immensely with debugging, and he contributed several pieces of code including the routine to size memory, an early version of the communication ports, and modifications to defer the clock.

Several colleagues provided valuable suggestions. Peter Denning first suggested the use of layering. He gave me a draft of Denning *et. al.* [1981] and provided pointers to the other literature, all of which influenced the design.

Tom Murtagh helped work through several layering details.

Janice Cuny, Brian Kernighan, Edmund Lien, and Jacobo Valdes, all commented on an early draft. Bob Brown and Ran Ginosar provided especially helpful suggestions on later versions.

I thank my wife, Chris, for patiently reading many drafts for technical accuracy and syntactic correctness.

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. Several of them came from the UNIX Time-Sharing System developed at Bell Laboratories (see Ritchie and Thompson [1974]). Readers familiar with UNIX should be aware, however, that although many of the ideas, techniques, and names come from UNIX, the two systems are quite different internally – programs written for one system do not usually run on the other.

Finally, I am indebted to Purdue University and Bell Laboratories for support of the project.

Douglas Comer

August 15, 1983