1

Introduction and Overview

1.1 Operating Systems

Hidden in every computer system is the software that controls processing, manages resources, and communicates with external devices like disks and printers. Collectively, the programs that perform these chores are sometimes referred to as the *executive*, *monitor*, *task manager*, or *kernel*. We will use the broader term *operating system*.

Computer operating systems are among the most complex objects created by mankind: they allow multiple users to share the machine simultaneously, protect data from unauthorized access, and keep dozens of independent devices operating correctly. All this is done at blinding speed by issuing detailed commands to incredibly intricate hardware. But the operating system is not usually an independent machine that sits around all day controlling the computer – it is itself a program that is executed by the same processor that executes user's programs. When the computer is executing a user's program, the operating system is inactive.

Arranging details so the operating system will always regain control is complicated enough, but it is even more impressive that an operating system manages to provide reasonably high-level services with unreasonably low-level hardware. As this book proceeds, we will see how crude hardware is and how much system software it takes to manage even simple devices like terminals. The philosophy behind our design is that operating systems need not be confined by the hardware; they can hide the low-level details of the real machine and provide the high-level services of an abstract machine.

Operating system design is not an old craft; an understanding of it has evolved slowly along with our understanding of machine architectures. In the beginning, machines were scarce and expensive; only a few programmers had an opportunity to use them, and fewer still had an opportunity to build operating systems. Because the basic problems of concurrent computation and automated resource management had not been solved, com-

mercial systems often contained major design flaws. They were unnecessarily complex and riddled with errors. Their internal details varied from machine to machine because they were intimately concerned with hardware resources. Their external capabilities and human interfaces also varied widely as vendors sought new ways to attract customers.

As technology grew, machines became less costly and more abundant. Advances in microelectronic technology reduced fabrication costs and made possible inexpensive single-board and single-chip computers. Vendors now offer customized chips. Designing and implementing software systems for microcomputers is no longer a task reserved for a few specialists; it is a skill expected of competent systems programmers.

Fortunately, our understanding of operating system design has grown along with the technology used to produce new machines. As researchers examined fundamental issues and experimented with system design, they began to formulate principles and techniques of good design. They identified the abstract services common to all operating systems and explored the many variations possible. They defined the basic operating system components that carry out these machine-independent abstractions, and discovered a technique called *layering* that organizes the components, simplifies system design, and eases implementation.

Compared to its early counterpart, a modern system is simple, clean, and portable. A modern system is easier to design because it follows a pattern, and easier to understand or modify because it is cleanly partitioned into layers. At the heart of the layered organization is the raw machine. Building out from this core, higher layers of software provide more powerful primitives and shield the user from the machine beneath. Each layer of the system provides an abstract service, implemented in terms of the abstract services provided by lower level layers.

1.2 Our Approach

This book is a guide to the design and implementation of layered operating systems. It takes a practical approach, showing the details of a real system. We begin with a microcomputer, an easy machine to understand, and proceed step-by-step through the construction of a layered system capable of supporting multiple processes, network communication, and a file system. Each chapter explains the role of one layer in the system, illustrating the details with programs. The design ends with a complete, working system that fits together in a clean and elegant way. The resulting system is not a toy; it is a powerful operating system shoehorned into a microcomputer.

Our approach provides two advantages. First, every part of the system is present; the reader will see how an entire system fits together, not merely how one or two important parts interact. Because the code for every piece is included there can be no mystery about any part of the implementation. Second, the reader can obtain a copy of the system to modify, instrument, measure, extend, or transport to another architecture.

Learning from a real system means that the programs form an integral part of the text that cannot be ignored. They are the centerpiece of discussion and must be read and studied to appreciate the underlying subtlety and engineering detail. Many of the exer-

cises, for example, suggest improvements or modifications that require the reader to delve into details. A skillful programmer will find additional ways to improve the code.

The key to a successful design lies in ordering the layers, so services needed to implement a given layer are defined in layers beneath it. In practice, design is a trial-and-error process where the layering is reorganized as design proceeds. To eliminate futile attempts and the consequent backtracking, we can use the results of research and the accumulated experience of other designers. Instead of considering alternative organizations, we simply begin with a layering scheme that is known to work well and follow it consistently. The organization selected is a process-based layering scheme that is versatile and widespread. It has several other useful properties that will become apparent as well (e.g., successively larger subsets form successively more powerful systems). By the end of the book, the reader will see how each piece of the system fits into this scheme and be prepared to tackle alternative organizations.

1.3 What An Operating System Is Not

Before proceeding into the design of an operating system, we should all agree on what it is we are about to study. Surprisingly, many programmers do not even have a correct intuitive definition of an operating system. Perhaps the problem arises because vendors and computer professionals often use one name to refer to the set of all software supplied with an operating system as well as the system itself. Perhaps it arises because the available support software usually makes it unnecessary for programmers to access system services directly, or perhaps because the user interface gives the system its characteristic personality. In any case, we can clear the air quickly by ruling out well-known items that are not part of the operating system core.

First, an operating system is not a language or a compiler, even though vendors usually supply compilers with their systems. All programs, even the operating system, must be written in some language. Although recent languages like Concurrent Euclid aid in writing operating systems, we will see that a system can be constructed using a conventional language and a conventional compiler.

Second, an operating system is not merely a command interpreter, although most vendors supply a command interpreter as the human interface to their systems. In older systems, designers chose to build command interpreters that users could not replace easily, causing users to assume that command interpretation was inherently linked to the operating system. In modern systems, command interpreters are like other programs; individual users can choose one that suits them or write their own.

Third, an operating system is not a library of commands. Utility programs that edit files, send mail, compile programs, or link them are just that – utility programs that any competent programmer can write and run without changing the operating system. Vendors usually supply a set of utilities with their operating systems, but local installations often change or replace them without modifying the system itself. Modern systems extend the freedom to replace commands to users, so they can tailor the computing environment according to their individual tastes.

1.4 An Operating System Viewed From The Outside

The essence of an operating system lies in the services it provides to user programs. Programs access these services by making *system calls*. System calls look like procedure calls when they appear in a program, but transfer to operating system routines when invoked at run-time. Taken as a set, the system calls establish a well-defined boundary between the running program and the operating system. They define the services that the system provides and the interface to those services.

To appreciate the interior of an operating system, one must first understand the characteristics of the services it provides and how to use those services. This book describes an operating system called *PC-Xinu*. We will begin by reviewing a few of the services it provides. Later we will return and describe, in detail, the implementation of each.

PC-Xinu runs on a microcomputer along with code from the user's programs. It performs chores such as reading characters from a keyboard; displaying characters on a terminal; managing multiple, simultaneous computations; operating timers; saving files on disk storage devices; and relaying messages between programs. The description that follows will help you understand, in general terms, how to use PC-Xinu, and the examples that follow will help explain some of its services.

1.4.1 The PC-Xinu Small Machine Environment

Small computers, like the one described in this book, are themselves capable of compiling operating systems as large as the ones that control them. Such a system has enough speed, storage, and system software that it behaves like any other general-purpose computer system: the programmer can create programs, compile them, save them, and run them using only the microcomputer. In the past, small machines did not have enough software or storage resources for these activities, and compiling an operating system could take days or weeks.

PC-Xinu runs on the same microcomputer system which is used for its development and compilation. The development environment includes an editor, compiler, assembler, linker, and library management utilities which are used together to create a library of routines – the heart of PC-Xinu. When a user module is linked with this library and the resulting program is run, PC-Xinu springs to life as an operating system essentially independent of its host environment.

1.4.2 PC-Xinu Services

Programs running under PC-Xinu access services by calling operating system routines. For example, the system routine *putc* writes a single character on an I/O device. It takes two arguments: the device identifier and the character to write. Here is a C† program that writes the message ''hi'' on the console when run under PC-Xinu:

---

†Appendix 1 contains a quick introduction to C that should be sufficient to understand programs in this book.

```
/* ex1.c - xmain */

#include <conf.h>

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

The code introduces several conventions used in PC-Xinu and in this book. The statement, *#include <conf.h>* inserts a file of configuration declarations in the source program. The configuration file contains, among other things, a definition for *CONSOLE*. Usually, *CONSOLE* refers to a terminal connected to the micro through which the user interacts. Later we will see the contents of *conf.h* and learn how names like *CONSOLE* become synonymous with devices; all a user needs to know is that the include statement must appear in any program that uses device names.

The program writes three characters to the terminal: ‘‘h’’, ‘‘i’’, and a newline. Newline is a control character that moves the cursor to the beginning of the next line.

The source file also introduces an important convention followed throughout this book. It begins with a one-line comment that gives the name of the file, *ex1.c*. If a source file contains several procedures, their names all appear on this line. Knowing the names of files will help you locate them if you have a machine-readable copy of PC-Xinu. In addition, a comment of the form

```
/*------------------------------------------------------------------------
 *  name  --  description
 *------------------------------------------------------------------------
 */
```

precedes all procedure or program definitions to help highlight them.

1.4.3 Concurrent Processing

Conventional programs are called *sequential* because the programmer imagines a machine executing the code statement-by-statement. At any instant, the machine is executing exactly one statement. Operating systems support a much larger view of computation called *concurrent processing*. Concurrent processing means that many computations

proceed ‘‘at the same time.’’

It is not difficult to imagine several independent programs each being executed statement-by-statement on several machines, but it is exceedingly difficult to imagine several independent computations being performed on a processor that can execute only one instruction at a time. Is concurrent computation real or imagined? If it is real, how does the hardware keep each program from interfering with the others? How do the programs cooperate so that only one takes control of an input or output device at a given time?

Computer systems usually do have some concurrent capabilities, but the most visible form of concurrency, multiple independent programs executing simultaneously, is a grand illusion. To create the illusion, an operating system switches a single processor among multiple programs, allowing it to execute one for only a few thousandths of a second before moving on to another. When viewed by a human, the programs appear to proceed concurrently. The technique, called *multiprogramming*, appears in almost all commercial systems. Interactive *multiprogramming* systems are called *timesharing systems* if the policy used to switch the processor around gives all users equal amounts of CPU time. The chief characteristic of a timesharing system is that the service received by a single user is inversely proportional to the load on the system.

Multiprogramming is a misleading term because concurrent processing encompasses more than ‘‘many instances of conventional programs.’’ To be more accurate we should say that an operating system switches the CPU among many ‘‘computations’’ called *processes*, *jobs*, or *tasks*. Because terminology varies from system to system, it is difficult to choose a term that accurately reflects the notion of ‘‘computation’’ in PC-Xinu. The terms *process* or *job* usually connote an isolated computation, while *task* often refers to one of a set of cooperating computations. In particular, concurrent programming languages often use the term *task* to refer to computations that share memory.

PC-Xinu refers to ‘‘computations’’ as *processes*, the term used throughout the rest of this book. The next section helps distinguish the notion of ‘‘process’’ from the usual notion of ‘‘sequential program’’ by giving some examples. As we will see, this difference plays a central role in operating system design – everything must be built with it in mind.

1.4.4 The Distinction Between Programs And Processes

When programmers write a conventional (sequential) program, they imagine a single processor executing the program step-by-step without interruption or delay. However, a programmer writing code for concurrent processes must take an entirely different view. The operating system itself is a good example of a concurrent program. At any given instant, several processes (computations) may be executing. It may be that no two of them are executing the same program, but it may happen that they are all about to execute the same statement of one program. To further complicate matters, switching the processor among processes may cause one process to overtake another; no guarantees are made about their relative speeds. Designing the system procedures to operate correctly in a concurrent environment provides a tough intellectual challenge because they must all

cooperate no matter which system procedures the user processes call or in which order. An example will help explain the problem.

In PC-Xinu, a single process starts executing at the beginning of the user's main program *xmain* when the system begins. The initial process may continue execution by itself, or it may create new, independent processes. When one process creates a new one, the original continues to execute, and the new process begins executing concurrently. For example, the code from file *ex2.c* consists of a main program and two procedures, *prA* and *prB*.

```
/* ex2.c - xmain, prA, prB */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 *  xmain  --  example of creating processes in PC-Xinu
 *------------------------------------------------------------------------
 */
xmain()
{
        int prA(), prB();

        resume( create(prA, INITSTK, INITPRIO, "proc 1", 0) );
        resume( create(prB, INITSTK, INITPRIO, "proc 2", 0) );
}

/*------------------------------------------------------------------------
 *  prA  --  repeatedly print 'A' without ever terminating
 *------------------------------------------------------------------------
 */
prA()
{
        while (1)
                putc(CONSOLE, 'A');
}

/*------------------------------------------------------------------------
 *  prB  --  repeatedly print 'B' without ever terminating
 *------------------------------------------------------------------------
 */
prB()
{
        while (1)
                putc(CONSOLE, 'B');
```

```
}
```

The main program never calls either procedure directly. Instead, it calls two operating system procedures, *create* and *resume*, passing the addresses of *prA* or *prB* as the first argument (other arguments to *create* specify such things as the stack space needed, a scheduling priority, process name, the count of arguments for the process, and the process arguments). Each call to *create* forms a new process that will begin executing instructions at the address specified by its first argument. *Create* sets up the process, leaving it ready to run, but temporarily suspended. It returns the process id of the new process to its caller (in some languages *create* would be called a ''function'' instead of a procedure). The *process id* is an integer that identifies the created process so it can be referenced later. In the example, the main program passes the process id returned by *create* to *resume* as an argument. *Resume* starts (unsuspends) the process so it begins executing. The distinction between normal procedure calls and process creation is this:

> *A procedure call does not return until the called procedure completes. Create and resume return to the caller after starting the process, allowing execution of both the calling procedure and the named procedure to proceed concurrently.*

All PC-Xinu processes execute independently and concurrently. In the example, the first new process executes code in procedure *prA*, printing the letter `A´ continuously; the second executes code in procedure *prB*, printing the letter `B´ continuously. Because processes execute concurrently, the output is a mixture of `A´s and `B´s. What happens to the main program? Remember that each independent computation is a process, so we should ask, ''What happens to the process executing the main program?'' The process executing the main program exits after the second call to resume because it has reached the end of the main program. Its exit does not affect the new processes at all. They go on spewing out `A´s and `B´s forever.

The example below shows that independent processes need not execute independent code. Just as in the previous example, a single process begins executing the main program. It calls *create* twice to start two new processes; both execute the code from procedure *prntr*.

```
/* ex3.c - xmain, prntr */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 *  xmain  --  example of 2 processes executing the same code concurrently
 *------------------------------------------------------------------------
 */
```

```
xmain()
{
        int prntr();

        resume( create(prntr, INITSTK, INITPRIO, "print A", 1, 'A') );
        resume( create(prntr, INITSTK, INITPRIO, "print B", 1, 'B') );
}

/*------------------------------------------------------------------------
 *  prntr  --  print a character indefinitely
 *------------------------------------------------------------------------
 */
prntr( ch )
int ch;
{
        while (1)
                putc(CONSOLE, ch);
}
```

The two processes proceed concurrently without any effect on one another, even though they happen to be executing the same piece of code. The key point here is that the notion of *process* is different from the usual notions of *program*:

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

This gives us some hint of the difficulty involved in designing operating systems. Not only must each piece be designed to operate correctly by itself, the designer must also guarantee that it does not interfere with other pieces, no matter how many processes execute simultaneously.

Although processes share code, it is important that each one have at least some local variables. If every variable was shared by every process, chaos would result whenever two processes tried to execute the same code. One can imagine what would happen if two processes tried to use a single variable as the index of a *for* loop. To avoid such interference the system creates an independent set of local variables for each process.

Create even allocates an independent set of arguments for each process, as the example demonstrates. The code in file *ex3.c* shows how two processes are passed different arguments even though they execute the same code. In the call to *create*, the last two arguments specify a count of arguments that follow, and a character that the system passes to the newly created process. The first new process created is passed character `A´, so it begins execution with formal parameter *ch* set to `A´. The second new process begins with *ch* set to `B´. Although these processes execute the same code, they each have their own copy of *ch*, just as recursive invocations of a procedure have their own

copy of formal parameters. As in the earlier example, the output contains a mixture of both letters. This points out another significant difference between programs and processes:

> *Storage for local variables and procedure arguments is associated with the process executing the procedure, not with the code in which they appear.*

In terms of the implementation, each process has its own stack of local variables, formal parameters, and procedure calls.

1.4.5 Process Exit

The previous example consisted of a concurrent program with three processes: the initial process, and the two processes started with the system call *create*. We said that the initial process ceased when it reached the end of the code in the main program; this is referred to as *process exit*. Other processes can exit in the same way, namely, by reaching the end of the procedure in which they start (or by returning from it). Once the process exits, it disappears forever; there is simply one less computation in progress.

You should not confuse process exit with normal procedure call and return or with recursive procedure calls. Just like a sequential program, each process has its own stack of procedure calls. Whenever it executes a call, the called procedure is pushed onto the stack. Whenever it returns from a procedure, the procedure is popped off the stack. Process exit occurs only when the process pops the last procedure (or main program) off its stack.

The system routine *kill* provides another mechanism to terminate a process. In a sense, *kill* is the inverse of *create*. It takes a process id as an argument and stops that process immediately. A process can be killed at any time and at any level of procedure nesting. When terminated, the process ceases execution, and its local variables disappear. The entire record of local variables and procedure calls on the stack disappears as well, of course. A process can exit by killing itself as easily as it can kill another process. To do so, it uses system call *getpid* to obtain its own process id and *kill* to terminate:

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

When used in this manner, the call to *kill* never returns because the calling process exits.

1.4.6 Shared Memory

In PC-Xinu, each process has its own copy of local variables, formal parameters, and procedure 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. Sharing also introduces the need for more operating system services. Consider a simple example of two processes that want

to communicate through a shared integer, *n*.

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

#include <conf.h>
#include <kernel.h>

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

/*------------------------------------------------------------------------
 *  xmain  --  example of unsynchronized producer and consumer processes
 *------------------------------------------------------------------------
 */
xmain()
{
        int     produce(), consume();

        resume( create(consumer,INITSTK,INITPRIO,"cons",0) );
        resume( create(producer,INITSTK,INITPRIO,"prod",0) );
}

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

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

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

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

In the code, global variable *n* is a shared integer, initialized to zero. The process executing *produce* loops 2000 times, incrementing *n*; we call this process the *producer*. The process executing *consume* also loops 2000 times; it prints the value of *n* in decimal. We call this process the *consumer*.

1.4.7 Synchronization

Try running *ex4.c* under PC-Xinu – its output may surprise you. Most programmers suspect that the consumer will print most, 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 several lines, then possibly another value; after that, its value is 2000. Even though the two processes run concurrently, they do not require the same amount of time. The consumer process must format and write a line of output, operations that require 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 characters to be sent 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 most of its loop in the short time it takes the consumer process to print a few lines. After a few output operations, the consumer process finds that *n* has the value 2000.

Production and consumption of data by independent processes is common. The question arises, ‘‘How can the programmer synchronize producer and consumer processes so that the consumer receives every datum produced?’’ Clearly, the producer must wait for the consumer to access the datum. Likewise, the consumer must wait for the producer to manufacture it. However, the mechanism for synchronization must be designed carefully, because the crucial constraint is this:

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

A process that executes instructions while waiting for another is said to be *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 CPU time and wind up waiting forever.

PC-Xinu avoids busy waiting by supplying coordination primitives called *semaphores* and system calls that operate on them. Each semaphore consists of an integer value, initialized when the semaphore is created. The system call *wait* decrements a semaphore and causes the process to delay if the result is negative. *Signal* performs the opposite action, incrementing the semaphore and allowing a waiting process to continue. To synchronize with semaphores, the producer and consumer need two semaphores, one on which the consumer waits, and one on which the producer waits. Semaphores are created dynamically with the system call *screate*, which takes the desired initial count as an argument, and returns an integer by which the semaphore is known.

In the example below, 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 runs, the consumer prints all values of *n* from 0 through 1999.

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

#include <conf.h>
#include <kernel.h>

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

/*------------------------------------------------------------------------
 *  xmain  --  producer and consumer processes synchronized with semaphores
 *------------------------------------------------------------------------
 */
xmain()
{
        int     prod2(), cons2();
        int     produced, consumed;

        consumed = screate(0);
        produced = screate(1);
        resume(create(cons2,INITSTK,INITPRIO,"cons",2,consumed,produced));
        resume(create(prod2,INITSTK,INITPRIO,"prod",2,consumed,produced));
}

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

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

```
        }
}

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

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

1.4.8 Mutual Exclusion

Semaphores provide another important purpose in PC-Xinu, namely, mutual exclusion. Two or more processes engage in *mutual exclusion* when they cooperate so that only one of them obtains access to a resource at a given time. For example, suppose two executing processes each, from time to time, want to insert an item into a linked list. If they both happen to access the list concurrently, they could leave pointers set incorrectly. Synchronization is not the answer because the two processes do not want to alternate accesses; they merely want to exclude each other.

To use mutual exclusion for control of a resource like a linked list, the processes must create a semaphore (with an initial count of 1). Before accessing the resource, each process executes *wait* on the semaphore and calls *signal* after it has completed. Often, the calls to *wait* and *signal* can be imbedded at the beginning and end of a procedure designed to perform the update. For example, file *ex6.c* shows an array and a procedure to add items to it:

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

int     mutex;                  /* assume initialized with screate      */
int     a[100];
int     n = 0;

/*------------------------------------------------------------------------
 *  additem  --  obtain exclusive access to array 'a' and add item to it
```

```
 *---------------------------------------------------------------------
 */
additem(item)
{
        wait(mutex);
        a[n++] = item;
        signal(mutex);
}
```

The assumption here is that a process created semaphore *mutex* using *screate* before any process called *additem*.

The code in file *ex6.c* provides a final illustration of the difference between the way one writes programs in sequential and concurrent environments. In the world of sequential programs, a procedure often acts to isolate changes to a data structure. By localizing code that changes a data structure in one procedure, the programmer gains a sense of security – only a small amount of code need be checked for correctness because nothing else in the program will interfere with the data structure. In a multiprocess environment, however, isolating the code into a single procedure is insufficient. The programmer must also guarantee that its execution is exclusive because interference can come from some other process executing the same procedure!

1.5 An Operating System Viewed From The Inside

If designed well, the interior of an operating system can be as elegant and clean as the best sequential program. The design described in this book achieves elegance by partitioning the system functions into roughly eight major components and organizing those components into a layered hierarchy. Examining this particular system is especially helpful in understanding layered organization because it demonstrates how all the functions in a conventional operating system fit together.

Figure 1.1 shows an overview of the components we will discuss, as well as their ultimate organization. Although the operating system structure is shown in final form, it was designed one layer at a time. At the heart of the system lies the scheduler and context switch. They are responsible for switching the CPU among the processes that are ready to run. Procedures in the next layer constitute the rest of the process manager, providing primitives to create, kill, suspend, and resume processes, and to manage memory.

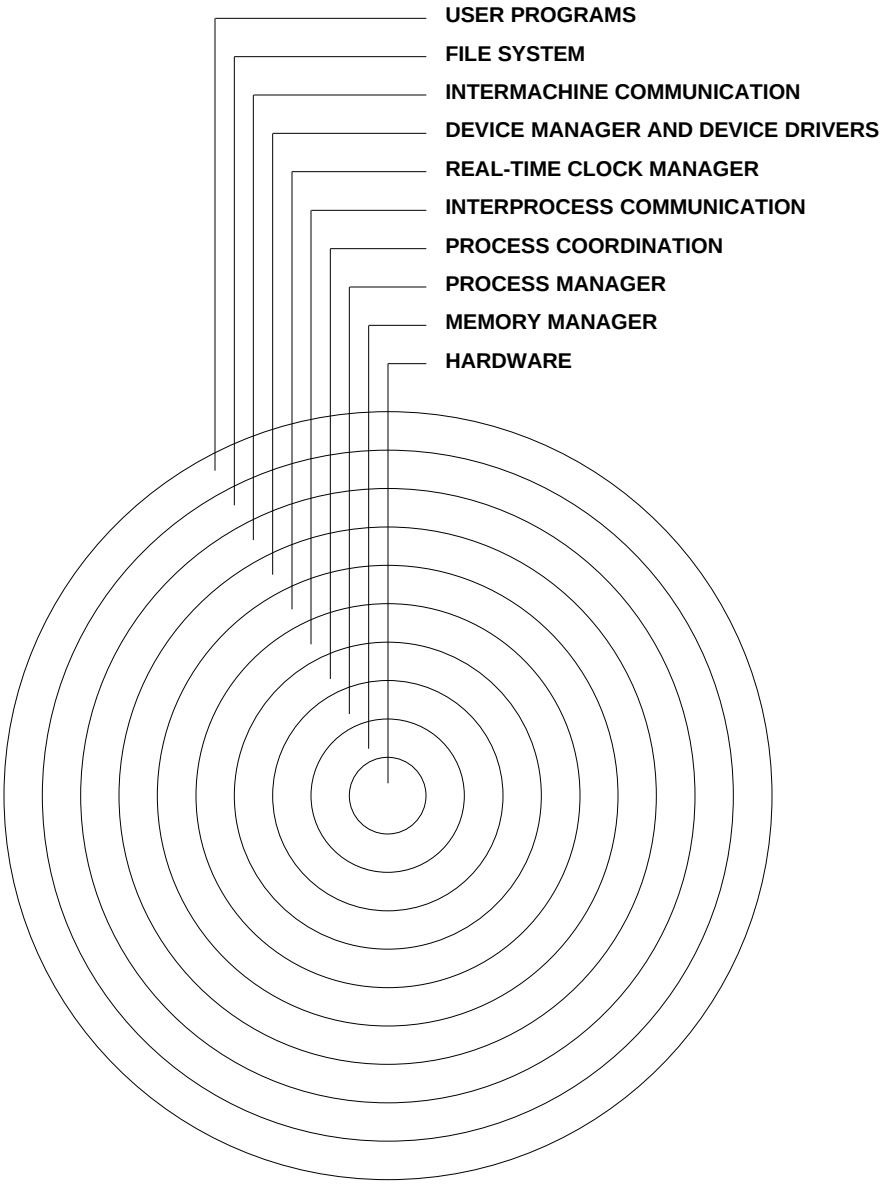

Figure 1.1 The layering of components in PC-Xinu

Just beyond the process management layer comes the process coordination layer that im-

plements semaphores. Following that comes the layer that provides message passing. Next come the procedures for real-time clock management that allow, among other things, processes to delay for a specified time. On top of the real-time clock layer lies a layer of device-independent input and output routines. Above the device routines, a layer implements machine-to-machine communication, and the layer above that implements a file system. This text does not include the network communication code.

The internal layering of a system should not be confused with the services it provides. The components are organized into layers to make the design and implementation cleaner; layering does not restrict procedure calls at run-time. Once the system has been built, procedures in higher layers can call routines like *wait* and *signal* that reside in the process coordination layer directly, just as they can call routines like *putc* that reside in outer layers. Thus, the layered structure describes only the implementation, not the flow of control.

The remainder of this book proceeds through the design of a system that follows the layered organization shown above. We consider the layers in roughly the same order as they are designed and built, from the innermost outward. Although this may seem awkward at first, the organization should start to seem meaningful by Chapter 6. By the end of Chapter 13, we will have designed a minimal kernel capable of supporting programs like those in the examples above. The design will include a complete core of operating system routines (including windowing and a file system) by the end of Chapter 18.

1.6 Summary

Operating system design has become a task expected of systems programmers. Unlike conventional, sequential programs, operating systems support a broad notion of computation in which multiple, independent processes execute concurrently and provide high-level abstract services. To design operating systems intelligently, programmers must appreciate concurrent processing and the services like mutual exclusion and process synchronization needed to support it.

The examples given here show how programs use a few of the basic operating system services. They illustrate the differences between sequential programs and concurrently executing processes, and show how processes begin and end in PC-Xinu. The examples illustrating the use of semaphores show how processes synchronize, and how they cooperate to guarantee mutual exclusion.

This introduction has focused on the exterior of an operating system, showing how programs use the services it provides. Later chapters concentrate on the interior of an operating system, showing how to design and build it, instead of how to use it. They proceed through the system one layer at a time, beginning with the raw hardware, and ending with a working system.

FOR FURTHER STUDY

Good surveys of operating system facilities can be found in Calingaert [1982], Habermann [1976], Lister [1984], Peterson and Silberschatz [1983], and Tsichritzis and Bernstein [1974]. Warwick [1970] provides a somewhat older view. Books by Brinch Hansen [1973], Habermann [1976], Holt [1983], Bic and Shaw [1988], and Lister [1984] all consider design issues. Readers interested in more information will find that journal papers provide deeper treatments of the subject. Selected references can be found at the end of each chapter, and in the Bibliography.

Details about PC-Xinu system calls and library routines are given throughout the text. They are summarized in the manual in Appendix 2, which also describes the development environment.

EXERCISES

1.1 Explore the system calls available on your favorite operating system, and write a few programs that use them.

1.2 The program in file *ex3.c* has 3 processes running at one point. Modify it to use only 2 processes.

1.3 Under PC-Xinu, the output from the programs in files *ex2.c* and *ex3.c* usually consists of alternate A's and B's. Speculate about the implementation of *putc*. What happens if one of the processes runs at a higher priority than the other?

1.4 Test the program in file *ex4.c* repeatedly. Does it always print the same number of zeroes? If it prints a value of *n* other than 0 or 2000, is this value always the same?

1.5 Modify the producer-consumer code in file *ex5.c* to use a buffer of 15 integers. Have the producer write integers 1, 2, ... in successive locations of the buffer, wrapping around to the beginning after filling the last slot, and have the consumer read and print them. Do you appreciate how to use counting semaphores with a buffer of size $k > 1$?

1.6 In file *ex5.c*, the semaphore *consumed* is created with a count of 1. Modify the code so *consumed* is created with a count of 0, and have the producer wait on it *after* signalling *produced*. Does it affect the output?

1.7 Write programs to find out what happens to a process that executes *wait* on a nonexistent semaphore or an existing semaphore that no other process signals.