CHAPTER

Enforcing Modularity with Virtualization

5

CHAPTER CONTENTS

Principles of Computer System Design: An Introduction

DOI: 10.1016/B978-0-12-374957-4.00014-1

OVERVIEW

The goal of the client/service organization is to limit the interactions between clients and services to messages. To ensure that there are no opportunities for hidden interactions, the previous chapter assumed that each client module and service module runs on a separate computer. Under that assumption, the network between the computers enforces modularity. This implementation reduces the opportunity for programming errors to propagate from one module to another, but it is also good for achieving security (because the service module can be penetrated only by sending messages) and fault tolerance (service modules can be separated geographically, which reduces the risk that a catastrophic failure such as an earthquake or a massive power failure affects all servers that implement the service).

The main disadvantage of using one computer per module is that it requires as many computers as modules. Since the modularity of a system and its applications shouldn't be dictated by the number of computers available, this requirement is undesirable. If the designer decides to split a system or application into n modules and would like to enforce modularity between them, the choice of n should not be constrained by the number of computers that happen to be in stock and easily obtained. Instead, the designer needs a way to run several modules on the same computer without resorting to soft modularity.

This chapter introduces *virtualization* as the primary approach to achieve this goal and presents three new abstractions (SEND and RECEIVE with *bounded buffers*, *virtual memory*, and *threads*) that correspond to virtualized versions of the three main abstractions (communication links, memory, and processors). The three new abstractions allow a designer to implement as many virtual computers as needed for running the desired *n* modules.

5.1 CLIENT/SERVER ORGANIZATION WITHIN A COMPUTER USING VIRTUALIZATION

To enforce modularity between modules running on the same computer, we create several *virtual* computers using one *physical* computer and execute each module (usually an application or a subsystem) in its own virtual computer.

This idea can be realized using a technique called *virtualization.* A program that virtualizes a physical object simulates the interface of the physical object, but it creates many virtual objects by *multiplexing* one physical instance, or it may provide one large virtual object by *aggregating* many physical instances, or implement a virtual object from a different kind of physical object using *emulation*. For the user of the simulated object, it provides the same behavior as a physical instance, but it isn't the physical instance, which is why it is called virtual. A primary goal of virtualization is to preserve an existing interface. That way, modules designed to use a physical instance of an object don't have to be modified to use a virtual instance. Figure 5.1 gives some examples of the three virtualization methods, which we will discuss in turn.

Hosting several Web sites on a single physical server is an example of virtualization involving multiplexing. If the aggregate peak load of the Web sites is less than what a

Virtualization Method	Physical Resource	Virtual Resource
multiplexing	server	Web site
multiplexing	processor	thread
multiplexing	real memory	virtual memory
multiplexing and emulation	real memory and disk	virtual memory with paging
multiplexing	wire or communication channel	virtual circuit
aggregation	communication channel	channel bonding
aggregation	disk	RAID
emulation	disk	RAM disk
emulation	Macintosh	virtual PC

FIGURE 5.1
Examples of virtualization.

single server computer can support, providers often prefer to use a single server to host several Web sites because it is less expensive than buying one server for each Web site.

The next three examples relate to threads and virtual memory, which we will overview in Section 5.1.1. Some of these usages don't rely on a single method of virtualization but combine several or use different methods to obtain different properties. For example, virtual memory with paging (described in Section 6.2) uses both multiplexing and aggregation.

A virtual circuit virtualizes a wire or communication channel using multiplexing. For example, it allows several phone conversations to take place over a single wire with a technique called time division multiplexing, as we will discuss in Chapter 7 [on-line]. Channel bonding aggregates several communication channels to provide a combined high data rate.

RAID (see Section 2.1.1.4) is an example of virtualization involving aggregation. In RAID, a number of disks are aggregated together in a clever way that provides an identical interface to the one of a single disk, but together the disks provide improved performance (by reading and writing disks concurrently) and durability (by writing information on more than one disk). A system administrator can replace a single disk with a RAID and take advantage of the RAID improvements without having to change the file system.

A RAM disk is an example of virtualization involving emulation. A RAM disk provides the same interface as a physical disk but stores blocks in memory instead of on a disk platter. RAM disks can therefore read and write blocks much faster than a physical disk but, because RAM is volatile, it provides little durability. Administrators can configure a file system to use a RAM disk instead of a physical disk without needing to modify the file system itself. For example, a system administrator may configure the file system to use RAM disk to store temporary files, which allows the file system to read and write temporary files fast. And since temporary files don't have to be stored durably, nothing is lost by storing them on a RAM disk.

A virtual PC is an example of virtualization using emulation. It allows the construction of a virtual personal computer out of a physical personal computer, perhaps of a different type (e.g., using a Macintosh to emulate a virtual PC). Virtual PCs can be useful to run several operating systems on a single computer, or simplify the testing and the development of a new operating system. Section 5.1.2 discusses this virtualization technique in more detail.

Designers are often tempted to tinker slightly with an interface rather than virtualizing it exactly, to improve it or to add a useful feature. Such tinkering can easily cross a line in which the original goal of not having to modify other modules that use the interface is lost. For example, the X Window System described in Sidebar 4.4 implements objects that could be thought of as virtual displays, but because the size of those objects can be changed on the fly and the program that draws on them should be prepared to redraw them on command, it is more appropriate to call them "windows".

Similarly, a file system (see Section 2.3.2) creates objects that store bits and thus has some similarity to a virtualized hard disk, but because files have names, are of adjustable length, allow controlled sharing, and can be organized into hierarchies, they are more appropriately thought of as a different memory abstraction.

The preceding examples suggest how we could implement the client/service organization within a single computer. Consider a computer on which we would like to run five modules: a text editor, an e-mail reader, a keyboard manager, the window service, and the file service. When a user works with the text editor, keyboard input should go to the editor. When the user moves the mouse from the editor window to the mail reader window, the next keyboard input should go to the mail reader. When the text editor saves a file, the file service must execute to store the file. If there are more modules than computers, some solution is needed for sharing a single computer.

The idea is to present each module with its own virtual computer. The power of this idea is that programmers can think of each module independently. From the programmer's perspective, every program module has a virtual computer to itself, which executes independently of the virtual computers of other modules. This idea enforces modularity because a virtual computer can contain a module's errors and no module can halt the progress of other modules.

The virtual computer design does not enforce modularity as well as running modules on physically separate computers because, for example, a power failure will knock out all virtual computers on the same physical computer. Also, once an attacker has broken into one virtual computer, the attacker may discover a way to exploit a flaw in the implementation of virtualization to break into other virtual computers. The primary modularity goal of employing virtual computers is to ensure that module failures due to accidental programming errors don't propagate from one virtual computer to another. Virtual computers can contribute to security goals but are better viewed as only one of several lines of defense.

5.1.1 Abstractions for Virtualizing Computers

The main challenge in implementing virtual computers is finding the right abstractions to build them. This chapter introduces three abstractions that correspond to virtualized versions of the main abstractions: SEND and RECEIVE with bounded buffers (virtualizes communication links), virtual memory (virtualizes memory), and threads (virtualizes processors).

These three abstractions are typically implemented by a program that is called the operating system (which was briefly discussed in Sidebar 2.4 but will be discussed in detail in this chapter). Using an operating system that provides the three abstractions, we can implement the client/service organization within a single computer (see Figure 5.2). For example, with this design the text editor running on one virtual computer can send a message over the virtual communication link to the file service, running on a different virtual computer, and ask it, for example, to save a file. In the figure each virtual computer has one virtual processor (implemented by a thread) and its own virtual memory with a virtual address space ranging from 0 to 2^n. To build an intuition for these abstractions and learn how they can be used to implement a virtual computer, we give a brief overview of them.

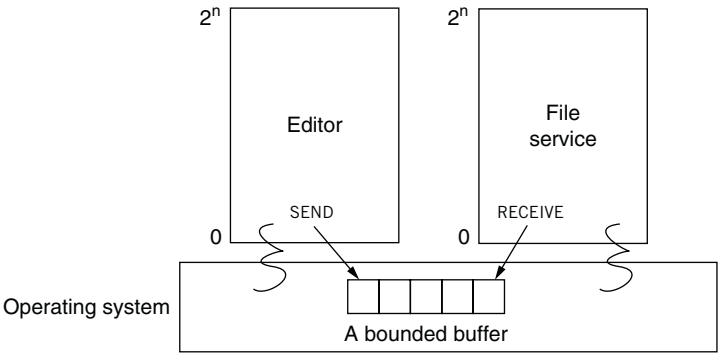


FIGURE 5.2

An operating system providing the editor and file service module each their own virtual computer. Each virtual computer has a thread that virtualizes the processor. Each virtual computer has a virtual memory that provides each module with the illusion that it has its own memory. To allow communication between virtual computers, the operating system provides SEND, RECEIVE, and a bounded buffer of messages.

5.1.1.1 Threads

The first step in virtualizing a computer is to virtualize the processor. To provide the editor module (shown in Figure 5.3) with a virtual processor, we create a *thread of execution*, or *thread* for short. A thread is an abstraction that encapsulates the execution state of an active computation. It encapsulates the state of a conceptual interpreter that executes the computation (see Section 2.1.2). The state of a thread consists of the variables internal to the interpreter (e.g., processor registers), which include

1. A reference to the next program step (e.g., a program counter)
2. References to the environment (e.g., a stack, a heap, and other current objects)

The thread abstraction encapsulates enough of the interpreter's state that one can stop a thread at any point in time, and later resume it. The ability to stop a thread and resume it later allows virtualization of the interpreter and provides a convenient way of multiplexing a physical processor. Threads are the most widely used implementation strategy to virtualize physical processors. In fact, this implementation strategy is so common that in the context of virtualizing physical processors the words "thread" and "virtual processor" have become synonyms in practice.

The next few paragraphs give a high-level overview of how threads can be used to virtualize physical processors. A user might type the name of the module that the user wants to run, or a user might select the name from a pull-down menu. The command line interpreter or the window system can then start the program as follows:

1. Load the program's text and data stored in the file system into memory.
2. Allocate a thread and start it at a specified address. Allocating a thread involves allocating a stack to allow the thread to make procedure calls, setting the SP register to the top of the stack, and setting the PC register to the starting address.

```
1    input ← OPEN (keyboard)          // open the keyboard device
2    file ← OPEN (argument)           // open the file that was passed an argument to the editor
3    do forever
4        n ← READLINE (input, buf)    // read characters from the keyboard into buf
5        APPLY (file, buf, n)         // apply them to the file being edited
```

FIGURE 5.3

Sketch of the program for the editor module.

A module may have one or several threads. A module with only *one* thread (and thus one processor) is common because then the programmer can think of it as executing a program serially: it starts at the beginning, computes (perhaps producing some output by performing a remote procedure call to a service), and then terminates. This simple structure follows the ***principle of least astonishment*** for programmers. Humans are better at understanding serial programs than at understanding programs that have several, concurrent threads, which can have surprising behavior.

Modules may have more than one thread by creating several threads. A module, for example, may create a thread per device that the module manages so that the module can operate the devices concurrently. Or a module may create several threads to overlap the latency of an expensive operation (e.g., waiting for a disk) by running the expensive operation in another thread. A module may allocate several threads to exploit several physical processors that run each thread concurrently. A service module may create several threads to process requests from different clients concurrently.

The thread abstraction is implemented by a *thread manager*. The thread manager's job is to multiplex the possibly many threads on the limited number of physical processors of the computer, and in such a way that a programming error in one thread cannot interfere with the execution of another thread. Since the thread encapsulates enough of the state so that one can stop a thread at any point in time, and later resume it, the thread manager can stop a thread and allocate the released physical processor to another thread by resuming that thread. Later the thread manager can resume the suspended thread again by reallocating a physical processor to that thread. In this way, the thread manager can multiplex many threads across a number of physical processors. The thread manager can ensure that no thread hogs a physical processor by forcing each thread to periodically give up its physical processor on a clock interrupt.

With the introduction of threads, it is helpful to refine the description of the interrupt mechanism described in Chapter 2. External events (e.g., a clock interrupt or a magnetic disk signals the completion of an I/O) interrupt a physical processor, but the event may have nothing to do with the thread running on the physical processor. On an interrupt, the processor invokes the interrupt handler and after returning from the handler continues running the thread that was running on the physical processor before the interrupt. If one processor should not be interrupted because it is already busy processing an interrupt, the next interrupt may interrupt another processor in the computer, allowing interrupts to be processed concurrently.

Some interrupts do pertain to the currently running thread. We shall refer to this class of interrupts as *exceptions.* The exception handler runs in the context of the interrupted thread; it can read and modify the interrupted thread's state. Exceptions often happen when a thread performs some operation that the hardware cannot complete (e.g., divide by zero). Many programming languages also have a notion of an exception; for example, a square root program may signal an exception if its caller hands it a negative argument. We shall see that because exception handlers run in the context of the interrupted thread, but interrupt handlers run in the context of the operating system, there are different restrictions on what the two kinds of handlers can safely do.

5.1.1.2 Virtual Memory

As described so far, all threads and handlers share the same physical memory. Each processor running a thread sends READ and WRITE requests across a bus along with an address identifying the memory location to be read or written. Sharing memory has benefits, but uncontrolled sharing makes it too easy to make a mistake. If several threads have their programs and data stored in the same physical memory, then the threads of each module have access to every other module's data. In fact, a simple programming error (e.g., the program computes the wrong address) can result in a STORE instruction overwriting another module's data or a JMP instruction executing procedures of another module. Thus, without a memory enforcement mechanism we have, at best, soft modularity. In addition, the physical memory and address space may be too small to fit the applications, requiring the applications to manage the memory carefully.

To enforce modularity, we must ensure that the threads of one module cannot overwrite the data of another module by accident. To do so, we give each module its own *virtual memory*, as Figure 5.2 illustrates. Virtual memory can provide each module with its own *virtual address space*, which has its own *virtual addresses*. That is, the arguments to JMP, LOAD, and STORE instructions are all virtual addresses, which a new hardware gadget (called a *virtual memory manager*) translates to physical addresses. If each module has its own virtual address space, then a module can name only its own physical memory and cannot store to the memory of another module. If a thread of a module by accident calculates an incorrect virtual address and stores to that virtual address, it will affect only that module.

With threads and virtual memory, we can create a virtual computer for each module. Each module has one or more threads that execute the code of the module. The threads of one module share a single virtual address memory that threads of other modules by default cannot touch.

5.1.1.3 Bounded Buffer

To allow client and service modules on virtual computers to communicate, we introduce SEND and RECEIVE with a *bounded buffer* of messages. A thread can invoke SEND, which attempts to insert the supplied message into a bounded buffer of messages. If the bounded buffer is full, the sending thread waits until there is space in the bounded buffer. A thread invokes RECEIVE to retrieve a message from the buffer; if there are no

messages, the calling thread waits. Using SEND, RECEIVE, and bounded buffers, we can implement remote procedure calls and enforce strong modularity between modules on different virtual computers running on the same physical computer.

5.1.1.4 Operating System Interface

To make the abstractions concrete, this chapter develops a minimal operating system that provides the abstractions (see Table 5.1 for its interface). This minimal design exhibits many of the mechanisms that are found in existing operating systems, but to keep the explanation simple it doesn't describe any existing system. Existing systems have evolved over many years, incorporating new ideas as they came along. As a result, few existing systems provide an example of a clean, simple design. In addition, a complete operating system includes many services (such as a file system, a window system, etc.) that are not included in the minimal operating system described in this chapter.

Table 5.1 The Interface Developed in this Chapter

Abstraction	Procedure
Memory	CREATE_ADDRESS_SPACE
	DELETE_ADDRESS_SPACE
	ALLOCATE_BLOCK
	FREE_BLOCK
	MAP
	UNMAP
Interpreter	ALLOCATE_THREAD
	EXIT_THREAD
	DESTROY_THREAD
	YIELD
	AWAIT
	ADVANCE
	TICKET
	ACQUIRE
	RELEASE
Communication	ALLOCATE_BOUNDED_BUFFER
	DEALLOCATE_BOUNDED_BUFFER
	SEND
	RECEIVE

5.1.2 Emulation and Virtual Machines

The previous section described briefly three high-level abstractions to virtualize processors, memory, and links to enforce modularity. An alternative approach is to provide an interface that is identical to some physical hardware. In this approach, one can enforce modularity by providing each application with its own instance of the physical hardware.

This approach can be implemented using a technique called *emulation.* Emulation simulates some physical hardware so faithfully that the emulated hardware can run any software the physical hardware can. For example, Apple Inc. has used emulation successfully to move customers to new hardware designs. Apple used a program named Classic to emulate Motorola Inc.'s 68030 processor on the PowerPC processor and more recently used a program named Rosetta to emulate the PowerPC processor on Intel Inc.'s x86 processor. As another example, some processors include a microcode interpreter inside the processor to simulate instructions of other processors or instructions from older versions of the same processor. It is also standard practice for a vendor developing a new processor to start by writing an emulator for it and running the emulator on some already existing processor. This approach allows software development to begin before the chip for the new processor is manufactured, and when the chip does become available, the emulator acts as a kind of specification against which to debug the chip.

Emulation in software is typically slow because interpreting the instructions of the emulated machine in software has substantial overhead. Looking at the structure of an interpreter in Figure 2.5, it is easy to see that decoding the simulated instruction, performing its operation, and updating the state of the simulated processor can take tens of instructions on the processor that performs the emulation. As a result, emulation in software can cost a factor 10 in performance and a designer must work hard to do better.

A specialized approach to fast emulation is using *virtual machines*. In this approach, a physical processor is used as much as possible to implement many virtual instances of itself. That is, virtual machines emulate many instances of a machine *M* using a physical machine *M*. This approach loses the portability of general emulation but provides better performance. The part of the operating system that provides virtual machines is often called a *virtual machine monitor*. Section 5.8 of this chapter discusses virtual machines and virtual machine monitors in more detail. Internally, a virtual machine monitor typically uses bounded buffers, virtual memory, and threads, the main topics of this chapter.

5.1.3 Roadmap: Step-by-Step Virtualization

This chapter gradually develops the tools needed to provide a virtual computer. We start out assuming that there are more physical processors than threads and that the operating system can allocate each thread its own physical processor. We will even assume that each interrupt handler has its own physical processor, so when

an interrupt occurs, the handler runs on that dedicated processor. Figure 5.4 shows a modified version of Figure 2.2, in which each thread has its own processor. Consider again the example that we would like to run the following five modules on a single computer: a text editor, an e-mail reader, a keyboard manager, the window service, and the file service. Processor 1, for example, might run the text editor thread. Processor 2 might run the e-mail reader thread. The window manager might have one thread per window, each running on a separate processor. Similarly, the file service might have several threads, each running on a separate processor. The LOAD and STORE instructions of threads refer to addresses that name memory locations or registers of the various controllers. That is, threads share the memories and controllers.

Given this setup, Section 5.2 shows how the client/server organization can be implemented in a computer with many processors and a single address space by allowing the threads of different modules to communicate through a bounded buffer. This implementation takes advantage of the fact that processors within a computer can interact with one another through a shared memory. That ability will prove useful to implement virtual communication links, but such unconstrained interaction through shared memory would drastically compromise modularity. For this reason, Section 5.3 will adjust this assumption to show how to provide and enforce walls between the memory regions used by different modules to restrict and control sharing of memory.

Sections 5.4, 5.5, and 5.6 of this chapter remove restrictions on the design presented in Sections 5.2 and 5.3. In Section 5.4 we will remove the restriction that processors must share one single, large address space, and provide each module with its own virtual memory, while still allowing controlled sharing. In Section 5.5 we remove the restriction that each thread must have its own physical processor while still ensuring that no thread can halt the progress of other threads involuntarily. Finally,

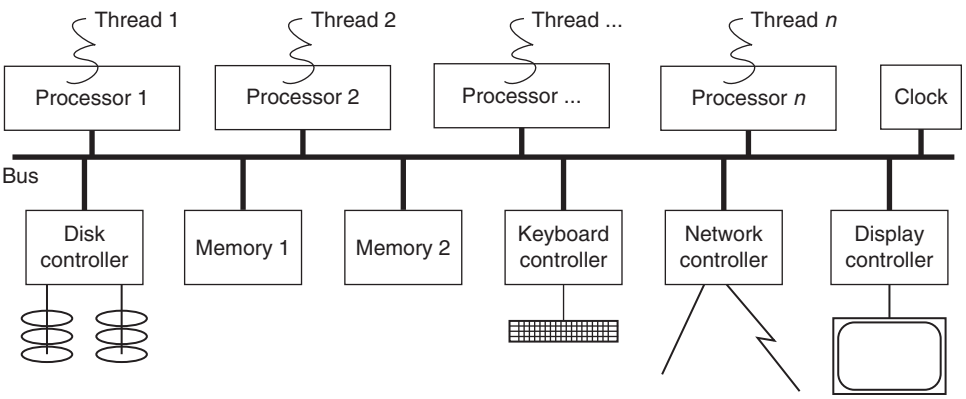


FIGURE 5.4

A computer with several hardware modules connected by a shared bus. Each thread of the software modules has its own processor allocated to it.

in Section 5.6 we remove the restriction that a thread must use a physical processor continuously to test if another thread has sent a message.

The operating system, thread manager, virtual memory manager, and SEND and RECEIVE with bounded buffers presented in this chapter are less complex than the designs found in contemporary computer systems. One reason is that most contemporary designs have evolved over time with changing technologies, while also allowing users to continue to run old programs. As an example of this evolution, Section 5.7 briefly describes the history of the Intel x86 processor, a widely used general-purpose processor design that has, over the years, provided increasing support for enforced modularity.

5.2 VIRTUAL LINKS USING SEND, RECEIVE, AND A BOUNDED BUFFER

Operating systems designers have developed many abstractions for virtual communication links. One popular abstraction is pipes [Suggestions for Further Reading 2.2.1 and 2.2.2], which allow two programs to communicate using procedures from the file system call interface. Because SEND and RECEIVE with a bounded buffer mirror a communication link directly, we describe them in more detail in this chapter. The implementation of SEND and RECEIVE with a bounded buffer also mirrors implementations of sockets, an interface for virtual links provided in operating systems such as UNIX and Microsoft Windows.

The main challenge in implementing SEND and RECEIVE with bounded buffers is that several threads, perhaps running in parallel on separate physical processors, may add and remove messages from the same bounded buffer concurrently. To ensure correctness, the implementation must coordinate these updates. This section will present bounded buffers in detail and introduce some techniques to coordinate concurrent actions.

5.2.1 An Interface for SEND and RECEIVE with Bounded Buffers

An operating system might provide the following interface for SEND and RECEIVE with bounded buffers:

- *buffer* ← ALLOCATE_BOUNDED_BUFFER (*n*): allocate a bounded buffer that can hold *n* messages.
- DEALLOCATE_BOUNDED_BUFFER (*buffer*): free the bounded buffer *buffer*.
- SEND (*buffer*, *message*): if there is room in the bounded buffer *buffer*, insert message in the buffer. If not, stop the calling thread and wait until there is room.
- *message* ← RECEIVE (*buffer*): if there is a message in the bounded buffer *buffer*, return the message to the calling thread. If there is no message in the bounded buffer, stop the calling thread and wait until another thread sends a message to buffer *buffer*.

SEND and RECEIVE with bounded buffers allow sending and receiving messages as described in Chapter 4. By building stubs that use these primitives, we can implement remote procedure calls between threads on the same physical computer in the same

way as remote procedure calls between physical computers. That is, from the client's point of view in Figure 4.8, there is no difference between sending a message to a local virtual computer or to a remote physical computer. In both cases, if the client or service module fails because of a programming error, then the other module needs to provide a recovery strategy, but it doesn't necessarily fail.

5.2.2 Sequence Coordination with a Bounded Buffer

The implementation with bounded buffers requires coordination between sending and receiving threads because a thread may have to wait until buffer space is available or until a message arrives. Two quite different approaches to thread coordination have developed over the years by researchers in different fields. One approach, usually taken by operating system designers, assumes that the programmer is an all-knowing genius who makes no mistakes. The other approach, usually taken by database designers, assumes that the programmer is a mere mortal, so it provides strong automatic support for coordination correctness, but at some cost in flexibility.

The next couple of subsections exhibit the genius approach to coordination, not because it is the best way to tackle coordination problems, but rather to give some intuition about why it requires a coordination genius, and thus should be subcontracted to such a specialist whenever possible. In addition, to implement the database approach the designer of the automatic coordination support approach must use the genius approach. Chapter 9 [on-line] uses the concepts introduced in this chapter to implement the database approach for mere mortals.

The scenario is that we have two threads (a sending thread and a receiving thread) that share a buffer into which the sender puts messages and the receiver removes those messages. For clarity we will assume that the sending and receiving thread each have their own processor allocated to them; that is, for the rest of this section we can equate thread with processor, and thus threads can proceed concurrently at independent rates. As mentioned earlier, Section 5.5 will explore what happens when we eliminate that assumption.

The buffer is bounded, which means that it has a fixed size. To ensure that the buffer doesn't overflow, the sending thread should hold off putting messages into the buffer when the number of messages there reaches some predefined limit. When that happens, the sender must wait until the receiver has consumed some messages.

The problem of sharing a bounded buffer between two threads is an instance of the *producer and consumer problem*. For correct operation, the consumer and the producer must coordinate their activities. In our example, the constraint is that the producer must first add a message to the shared buffer before the consumer can remove it and that the producer must wait for the consumer to catch up when the buffer fills up. This kind of coordination is an example of *sequence coordination*: a coordination constraint among threads stating that, for correctness, an event in one thread must precede an event in another thread.

Figure 5.5 shows an implementation of SEND and RECEIVE using a bounded buffer. This implementation requires making some subtle assumptions, but before diving

```
1     shared structure buffer                    // A shared bounded buffer
2         message instance message[N]            // With a maximum of N messages
3         integer in initially 0                 // Counts number of messages put in the buffer
4         integer out initially 0                // Counts number of messages taken out of the buffer

5     procedure SEND (buffer reference p, message instance msg)
6         while p.in − p.out = N do nothing      // If buffer is full, wait for room
7         p.message [p.in modulo N] ← msg         // Put message in the buffer
8         p.in ← p.in + 1                        // Increment in

9     procedure RECEIVE (buffer reference p)
10        while p.in = p.out do nothing          // If buffer is empty, wait for message
11        msg ← p.message [p.out modulo N]       // Copy item out of buffer
12        p.out ← p.out + 1                      // Increment out
13        return msg                             // Return message to receiver
```

FIGURE 5.5

An implementation of a SEND and RECEIVE using bounded buffers.

into these assumptions let's first consider how the program works. The two threads implement the sequence coordination constraint using N (the size of the shared bounded buffer) and the variables *in* (the number of items produced) and *out* (the number of items consumed). If the buffer contains items (i.e., *in* > *out* on line *10*), then the receiver can proceed to consume the items; otherwise, it loops until the sender has put some items in the buffer. Loops in which a thread is waiting for an event without giving up its processor are called *spin loops*.

To ensure that the sender waits when the buffer is full, the sender puts new items in the buffer only if *in* − *out* < N (line *6*); otherwise, it spins until the receiver made room in the buffer by consuming some items. This design ensures that the buffer does not overflow.

The correctness of this implementation relies on several assumptions:

1. The implementation assumes that there is one sending thread and one receiving thread and that only one thread updates each shared variable. In the program only the receiver thread updates *out,* and only the sender thread updates *in*. If several threads update the same shared variable (e.g., multiple sending threads update *in* or the receiving thread and the sending thread update a variable), then the updates to the shared variable must be coordinated, which this implementation doesn't do.

   This assumption exemplifies the principle that coordination is simplest when each shared variable has just one writer:

One-writer principle

If each variable has only one writer, coordination becomes easier.

That is, if you can, arrange your program so that two threads don't update the same shared variable. Following this principle also improves modularity because information flows in only one direction: from the single writer to the reader. In our implementation, *out* contains information that flows from the receiver thread to the sender, and *in* contains information that flows from the sender thread to the receiver. This restriction of information flow simplifies correctness arguments and, as we will see in Chapter 11 [on-line], can also enhance security.

A similar observation holds for the way the bounded buffer *buffer* is implemented. Because *messages* is a fixed-size array, the entries are written only by the sender thread. If the buffer had been implemented as a linked list, we might have a situation in which the sender and the receiver need to update a shared variable at the same time (e.g., the pointer to the head of the linked list) and then these updates would have to be coordinated.

2. The spin loops on lines *6* and *10* require the previously mentioned assumption that the sender and the receiver threads each run on a dedicated processor. When we remove that assumption in Section 5.5 we will have to do something about these spin loops.

3. This implementation assumes that the variables *in* and *out* are integers whose representation must be large enough that they will never overflow for the life of the buffer. Integers of width 64 or 96 bits would probably suffice for most applications. (An alternative way to remove this assumption is to make the implementation of the bounded buffer more complicated: perform all additions involving *in* and *out* modulo *N,* and reserve one additional slot in the buffer to distinguish a full buffer from an empty one.)

4. The implementation assumes that the shared memory provides read/write coherence (see Section 2.1.1.1) for *in* and *out*. That is, a LOAD of the variable *in* or *out* by one thread must be guaranteed to obtain the result of the most recent store to that variable by the other thread.

5. The implementation assumes before-or-after atomicity for the variables *in* and *out*. If these two variables fit in a 16- or 32-bit memory cell that can be read and written with a single LOAD or STORE, this assumption is likely to be true. But a 64- or 96-bit integer would probably require multiple memory cells. If they do, reading and writing *in* and *out* would require multiple LOADs or STOREs, and additional measures will be necessary to make these multistep sequences atomic.

6. The implementation assumes that the result of executing a statement becomes visible to other threads in program order. If an optimizing compiler or processor reorders statements to achieve better performance, this program could work incorrectly. For example, if the compiler generates code that reads *p.in* once, holds it in a temporary register for use in lines *6* through *8*, and updates the memory copy of *p.in* immediately, then the receiver may read the contents of the *in*th entry of the shared buffer before the sender has copied its message into that entry.

The rest of this section explains what problems occur when assumptions *1* (the one-writer principle) and *5* (before-or-after atomicity of multistep LOAD and STORE sequences) don't hold and introduces techniques to ensure them. In Section 5.5 we will find out how to remove assumption *2* (more processors than threads). Throughout, we assume that assumptions *3*, *4*, and *6* always hold.

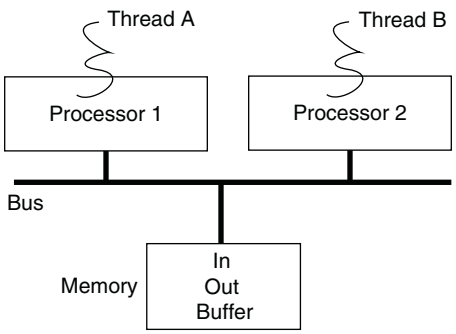
5.2.3 Race Conditions

To illustrate the importance of the six assumptions in guaranteeing the correctness of the program in Figure 5.5, let's remove two of those assumptions, one at a time, to see just what goes wrong. What we will find is that to deal with the removed assumptions we need additional mechanisms, mechanisms that Section 5.2.4 will introduce. This illustration reinforces the observation that concurrent programming needs the attention of specialists: all it takes is one subtle change to make a correct program wrong.

To remove the first assumption, let's allow several senders and receivers. This change will violate the one-writer principle, so we should not be surprised to find that it introduces errors. Multiple senders and receivers are common in practice. For example, consider a printer that is shared among many clients. The service managing the printer may receive requests from several clients. Each request adds a document to the shared buffer of to-be-printed documents. In this case, we have several senders (the threads adding jobs to the buffer) and one receiver (the printer).

As we will see, the errors that will manifest themselves are difficult to track down because they don't always show up. They appear only with a particular ordering of the instructions of the threads involved. Thus, concurrent programs are not only difficult to get right, but also difficult to debug when they are wrong.

The solution in Figure 5.5 doesn't work when several senders execute the code concurrently. To see why, let's assume N is 20 and that all entries in the buffer are empty (e.g., *out* is 0 and *in* is 0), and each thread is running on its own processor:



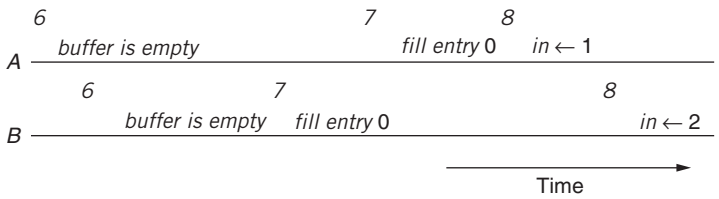
If two sending threads run concurrently—one on processor A and one on processor B—the threads issue instructions independently of each other, at their own

pace. The processors may have different speeds and take interrupts at different times, or instructions may hit in the cache on one processor and miss on another, so there is no way to predict the relative timing of the LOAD and STORE instructions that the threads issue.

This scenario is an instance of asynchronous interpreters (described in Section 2.1.2). Thus, we should make no assumptions about the sequence in which the memory operations of the two threads execute. When analyzing the concurrent execution of two threads, both executing instructions *6* through *8* in Figure 5.5, we can assume they execute in some *serial* sequence (because the bus arbiter will order any memory operations that arrive at the bus at the same time). However, because the relative speeds of the threads are unpredictable, we can make no assumptions about the order in the sequence.

We represent the execution of instruction *6* by thread A as "*A6*". Using this representation, we see that one possible sequence might be as follows: *A6, A7, A8, B6, B7, B8*. In this case, the program works as expected. Suppose we just started, so variables *in* and *out* are both zero. Thread *A* performs all of its three instructions before thread *B* performs any of its three instructions. With this order, thread *A* inserts an item in entry 0 and increments *in* from 0 to 1. Thread *B* adds an item in entry 1 and increments *in* from 1 to 2.

Another possible, but undesirable, sequence is *A6, B6, B7, A7, A8, B8*, which corresponds to the following timing diagram:



With this order, thread *A*, at *A6*, discovers that entry 0 of the buffer is free. Then, at *B6*, *B* also discovers that buffer entry 0 is free. At *B7*, *B* stores an item in entry 0 of *buffer*. Then, *A* proceeds: at *A7* it also stores an item in entry 0, overwriting *B*'s item. Then, both increment *in* (*A8* and *B8*), setting *in* first to 1 and then to 2. Thus, at the end of this order of instructions, one print job is lost (thread *B*'s job), and (because both threads incremented *in*) the receiver will find that entry 1 in the buffer was never filled in.

This type of error is called a *race condition* because it depends on the exact timing of two threads. Whether or not an error happens cannot be controlled. It is nasty, since some sequences deliver a correct result and some sequences deliver an incorrect result.

Worse, small timing changes between invocations might result in different behavior. If we notice that *B*'s print job was lost and we run it again to see what went wrong, we might get a correct result on the retry because the relative timing of

the instructions has changed slightly. In particular, if we add instructions (e.g., for debugging) on the retry, it is almost guaranteed that the timing is changed (because the threads execute additional instructions) and we will observe a different behavior. Bugs that disappear when the debugger starts to close in on them are colloquially called "Heisenbugs" in a tongue-in-cheek pun on the Heisenberg uncertainty principle of quantum mechanics. Heisenbugs are difficult to reproduce, which makes debugging difficult.

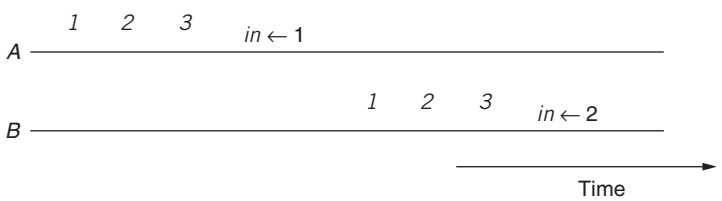
Race conditions are the primary pitfall in writing concurrent programs and the main reason why developing concurrent programs should be left to specialists, despite the existence of tools to help identifying races (e.g., see Savage et al. [Suggestions for Further Reading 5.5.6]). Concurrent programming is subtle. In fact, with several senders the program of Figure 5.5 has a second race condition. Consider the statement *8* that the senders execute:

$in \leftarrow in + 1$

In reality, a thread executes this statement in three separate steps, which can be expressed as follows:

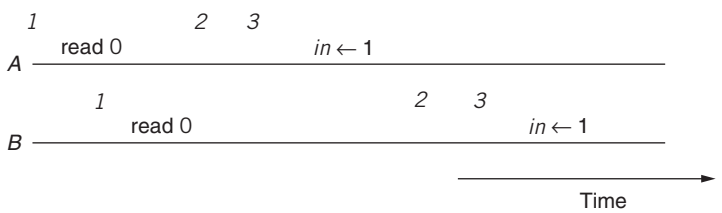
```
1  LOAD in, R0      // Load the value of in into a register
2  ADD R0, 1        // Increment
3  STORE R0, in     // Store result back to in
```

Consider two sending threads running simultaneously, threads *A* and *B*, respectively. The instructions of the threads might execute in the sequence *A1*, *A2*, *A3*, *B1*, *B2*, *B3*, which corresponds to the following timing diagram:



In this case *in* is incremented by two, as the programmer intended.

But now consider the execution sequence *A1*, *B1*, *A2*, *A3*, *B2*, *B3*, which corresponds to the following timing diagram:



When the two threads finish, this ordering of memory references has increased *in* by only 1. At *A1*, thread *A* loads the R0 register of its thread with the value of *in*, which is 0. At *B1*, thread *B* does exactly the same thing, loading its thread's register R0 with the value 0. Then, at *A2*, thread A computes the new value in R0 and at *A3* updates *in* with the value 1. Next, at *B2* and *B3*, thread *B* does the same thing: it computes the new value in R0 and updates *in* with the value 1. Thus, *in* ends up containing 1 instead of the intended 2. Any time two threads update a shared variable concurrently (i.e., the one-writer principle is violated), a race condition is possible.

We caused this second race condition by allowing multiple senders. But the manipulation of the variables *in* and *out* also has a potential race even if there is only one sender and one receiver, and we remove assumption *5* (the before-or-after atomicity requirement). Let's assume that we want to make *in* and *out* of type **long integer** so that there is little risk of overflowing those two variables. In that case, *in* and *out* each span two memory cells instead of one, and updates to *in* and *out* are no longer atomic operations. That change creates yet another race.

If *in* is a long integer, then updating *in* would require two instructions:

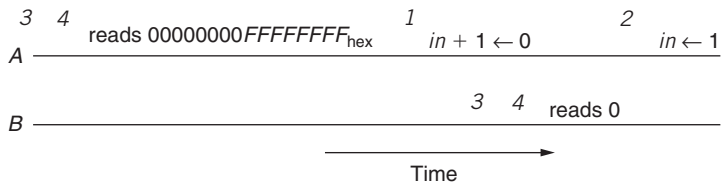
```
1  STORE R0, in+1    // Update the least-significant word of in
2  STORE R1, in      // Update the most-significant word of in
```

To read *in* would also require two instructions:

```
3  LOAD in+1, R0     // Load the least-significant word of in into a register
4  LOAD in, R1       // Load the most-significant word of in into a register
```

If the sender executes instructions *1* and *2* at about the same time that the receiver executes instructions *3* and *4*, a race condition could manifest itself. Let's assume that two threads call SEND and RECEIVE $2^{32}-1$ times, and interleave their calls perfectly. At this point there are no messages in the buffer and $in = out = 00000000FFFFFFFF_{hex}$ using big-endian notation.

Let's consider the scenario in which thread *A* has just added a message to the buffer, has read *in* into R0 and R1 (at instructions 3 and *4*), has computed the new value for *in* in the registers R0 and R1, and has executed instruction *1* to update *in* to memory. But before A executes instruction *2*, thread *B* adds a message:



In this case, the program works incorrectly because *A* has stored a message in entry 15 of the buffer ($00000000FFFFFFFF_{hex}$ **modulo** 20 = 15), *B* stores a message in entry 0, and *A* completes the update of *in*, which sets *in* to 0000000100000000_{hex}. *B*'s message in entry 0 will be lost because entry 0 will be overwritten by the next caller to SEND.

Race conditions are not uncommon in complex systems. Two notorious ones occurred in CTSS and in the Therac-25 machine. In CTSS, an early operating system, all running instances of a text editor used the same name for temporary files. At some point, two administrators were concurrently editing the file with the message of the day and a file containing passwords. The content of the two files ended up being exchanged (see Section 11.11.2 [on-line] for the details): when users logged into CTSS, it displayed the pass phrases of all other users as the message of the day.

The Therac-25 is a machine that delivers medical irradiation to human patients [Suggestions for Further Reading 1.9.5]. A race condition between a thread and the operator allowed an incorrect radiation intensity to be set: as a result, some patients died. The repairman could not reproduce the problem, since he typed more slowly than the more experienced operator of the machine.

Problem sets *4*, *5*, and *6* ask the reader to find race conditions in a few small, concurrent code fragments.

5.2.4 Locks and Before-or-After Actions

From the examples in the preceding section we can see that the program in Figure 5.5 was carefully written so that it didn't violate assumptions *1* and *5*. If we make slight modifications to the program or use the program in slightly different ways than it was intended to be used, we violate the assumptions and the program exhibits race conditions. We would like a technique by which a developer can systematically avoid race conditions. This section introduces a mechanism called a *lock*, with which a designer can make a multistep operation behave like a single-step operation. By using locks carefully, we can modify the program in Figure 5.5 so that it enforces assumptions *1* and *5*, and thus avoids the race conditions systematically.

A lock is a shared variable that acts as a flag to coordinate usage of other shared variables. To work with locks we introduce two new primitives: ACQUIRE and RELEASE, both of which take the name of a lock as an argument. A thread may ACQUIRE a lock, hold it for a while, and then RELEASE it. While a thread is holding a lock, other threads that attempt to acquire that same lock will wait until the first thread releases the lock. By surrounding multistep operations involving shared variables with ACQUIRE and RELEASE, the designer can make the multistep operation on shared variables behave like a single-step operation and avoid undesirable interleavings of multistep operations.

Figure 5.6 shows the code of Figure 5.5 with the addition of ACQUIRE and RELEASE invocations. The modified program uses only one lock (*buffer_lock*) because there is a single data structure that must protected. The lock guarantees that the program works correctly when there are several senders and receivers. It also guarantees correctness when *in* and *out* are long integers. That is, the two assumptions under which the program of Figure 5.5 is correct are now guaranteed by the program itself.

The ACQUIRE and RELEASE invocations make the reads and writes of the shared variables *p.in* and *p.out* behave like a single-step operation. The lock set by ACQUIRE and RELEASE ensures the test, and manipulation of the buffer is executed as one indivisible action; thus, no undesirable interleavings and races can happen. If two threads attempt

to execute the multistep operation between ACQUIRE and RELEASE concurrently, one thread acquires the lock and finishes the complete multistep operation before the other thread starts on the operation. The ACQUIRE and RELEASE primitives have the effect of dynamically implementing the one-writer principle on those variables: they ensure there is only a single writer at any instant, but the identity of the writer can change.

It is important to keep in mind that when a thread acquires a lock, the shared variables that the lock is supposed to protect are not mechanically protected from access by other threads. Any thread can still read or write those variables without acquiring the lock. The lock variable merely acts as a flag, and for correct coordination all threads must honor an additional convention: they must not perform operations on shared variables unless they hold the lock. If any thread fails to honor that convention, there may be undesirable interleavings and races.

To ensure correctness in the presence of concurrent threads, a designer must identify *all* potential races and carefully insert invocations of ACQUIRE and RELEASE to prevent them. If the locking statements don't ensure that multistep operations on shared variables appear as single-step operations, then the program may have a race condition. For example, if in the SEND procedure of Figure 5.6 the programmer places the ACQUIRE and RELEASE statements around just the statements on lines *11* through

```
1    shared structure buffer                    // A shared bounded buffer
2        message instance message[N]            // with a maximum of N messages
3        long integer in initially 0            // Counts number of messages put in the buffer
4        long integer out initially 0           // Counts number of messages taken out of the buffer
5        lock instance buffer_lock initially UNLOCKED // Lock to order sender and receiver

6    procedure SEND (buffer reference p, message instance msg)
7        ACQUIRE (p.buffer_lock)
8        while p.in − p.out = N do              // Wait until there is room in the buffer
9            RELEASE (p.buffer_lock)            // While waiting release lock so that RECEIVE
10           ACQUIRE (p.buffer_lock)            // can remove a message
11       p.message[p.in modulo N] ← msg         // Put message in the buffer
12       p.in ← p.in + 1                        // Increment in
13       RELEASE (p.buffer_lock)

14   procedure RECEIVE (buffer reference p)
15       ACQUIRE (p.buffer_lock)
16       while p.in = p.out do                  // Wait until there is a message to receive
17           RELEASE (p.buffer_lock)            // While waiting release lock so that SEND
18           ACQUIRE (p.buffer_lock)            // can add a message
19       msg ← p.message[p.out modulo N]        // Copy item out of buffer
20       p.out ← p.out + 1                      // Increment out
21       RELEASE (p.buffer_lock)
22       return msg
```

FIGURE 5.6

An implementation of SEND and RECEIVE that adds locks so that there can be multiple senders and receivers. The RELEASE and ACQUIRE on lines 9 and 10 are explained in Section 5.25.

12, then several race conditions may happen. If the lock doesn't protect the test of whether there's space in the buffer (line *8*), a buffer with only one space free could be appended to by multiple concurrent invocations to SEND. Also, before-or-after atomicity for *in* and *out* (assumption *5*) could be violated during the comparisons of *p.in* with *p.out*, so the race described in Section 5.2.3 could still occur. Programming with locks requires great attention to detail. Chapter 9 [on-line] will explore schemes that allow the designer to systematically ensure correctness for multistep operations involving shared variables.

A lock can be used to implement before-or-after atomicity. During the time that a thread holds a lock that protects one or more shared variables, it can perform a multistep operation on these shared variables. Because other threads that honor the lock protocol will not concurrently read or write any of the shared variables, from their point of view the multiple steps of the first thread appear to happen indivisibly: before the lock is acquired, none of the steps have occurred; after the lock is released all of them are complete. Any operation by a concurrent thread must happen either completely before or completely after the before-or-after atomic action.

The need for before-or-after atomicity has been realized in different contexts, and as a result that concept and before-or-after atomic actions are known by various names. The database literature uses the terms *isolation* and *isolated actions*; the operating system literature uses the terms *mutual exclusion* and *critical sections*; and the computer architecture literature uses the terms *atomicity* and *atomic actions*. Because Chapter 9 [on-line] introduces a second kind of atomicity, this text uses the qualified term "before-or-after atomicity" for precision as well as for its self-defining and mnemonic features.

In general, in the computer science community, a tremendous amount of work has been done on approaches to finding race conditions in programs and on approaches to avoid them in the first place. This text introduces the fundamental ideas in concurrent programming, but the interested reader is encouraged to explore the literature to learn more.

The usual implementation of ACQUIRE and RELEASE guarantees that only a single thread can acquire a given lock at any one time. This requirement is called the *single-acquire protocol*. If the programmer knows more details about how the protected shared variables will be used, a more relaxed protocol may be able to allow more concurrency. For example, Section 9.5.4 describes a multiple-reader, single-writer locking protocol.

In larger programs with many shared data structures, a programmer often uses several locks. For example, if each of the several data structures is used by different operations, then we might introduce a separate lock for each shared data structure. That way, the operations that use different shared data structures can proceed concurrently. If the program used just one lock to protect all of the data structures, then all operations would be serialized by the lock. On the other hand, using several locks raises the complexity of understanding a program by another notch, as we will see next.

Problem sets *4* and *5* explore several possible locations for ACQUIRE and RELEASE statements in an attempt to remove a race condition while still allowing for concurrent execution of some operations. Birrell's tutorial [Suggestions for Further Reading

5.3.1] provides a nice introduction on how to write concurrent programs with threads and locks.

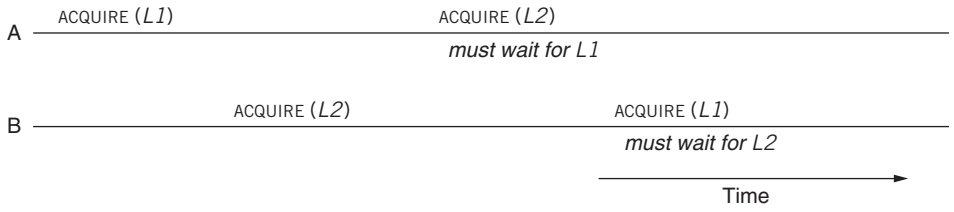
5.2.5 Deadlock

A programmer must use locks with care, because it is easy to create other undesirable situations that are as bad as race conditions. For example, using locks, a programmer can create a *deadlock*, which is an undesirable interaction among a group of threads in which each thread is waiting for some other thread in the group to make progress.

Consider two threads, A and B, that both must acquire two locks, *L1* and *L2*, before they can proceed with their task:

Thread A	Thread B
ACQUIRE(*L1*)	ACQUIRE(*L2*)
ACQUIRE(*L2*)	ACQUIRE(*L1*)

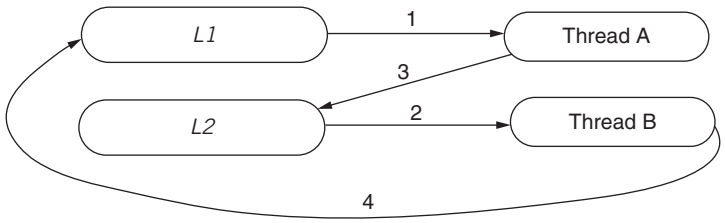
This code fragment has a race condition that results in deadlock, as shown in the following timing diagram:



Thread A cannot make forward progress because thread B has acquired *L2*, and thread B cannot make forward progress because thread A has acquired *L1*. The threads are in a deadly embrace.

If we had modified the code so that both threads acquire the locks in the same order (*L1* and then *L2*, or vice versa), then no deadlock could have occurred. Again, small changes in the order of statements can result in good or bad behavior.

A convenient way to represent deadlocks is using a *wait-for* graph. The nodes in a wait-for graph are threads and resources such as locks. When a thread acquires a lock, it inserts a directed edge from the lock node to the thread node. When a thread must wait for a resource, it inserts another directed edge from the thread node to the resource node. As an example, the race condition with threads A, B, and locks *L1* and *L2* results in the following wait-for graph:



When thread A acquires lock *L1*, it inserts arrow 1. When thread B acquires lock *L2*, it inserts arrow 2. When thread A must wait for lock *L2*, it inserts arrow 3. When thread B attempts to acquire lock *L1* but must wait, it inserts arrow 4. When a thread must wait, we check if the wait-for graph contains a cycle. A cycle indicates deadlock: everyone is waiting for someone else to release a resource. In general, if, and only if, a wait-for graph contains a cycle, then threads are deadlocked.

When there are several locks, a good programming strategy to avoid deadlock is to enumerate all lock usages and ensure that all threads of the program acquire the locks in the same order. This rule will ensure there can be no cycles in the wait-for graph and thus no deadlocks. In our example, if thread B above did ACQUIRE(*L1*) before ACQUIRE (*L2*), the same order that thread A used, then there wouldn't have been a problem. In our example program, it is easy for the programmer to modify the program to ensure that locks are acquired in the same order because the ACQUIRE statements are shown next to each other and there are only two locks. In a real program, however, the four ACQUIRE statements may be buried deep inside two separate modules that threads A and B happen to call indirectly in different orders, and ensuring that all locks are acquired in a static global order requires careful thinking and design.

A deadlock doesn't always have to involve multiple locks. For example, if the sender forgets to release and acquire the lock on lines *9* and *10* of Figure 5.6, then a deadlock is also possible. If the buffer is full, the receiver will not get a chance to remove a message from the buffer because it cannot acquire the lock, which is being held by the sender. In this case, the sender is waiting on the receiver to change the value of *p.out* (in a wait-for graph, the resource is buffer space represented by the value of *p.out*), and the receiver is waiting on the sender to release the lock. Simple programming errors can lead to deadlocks.

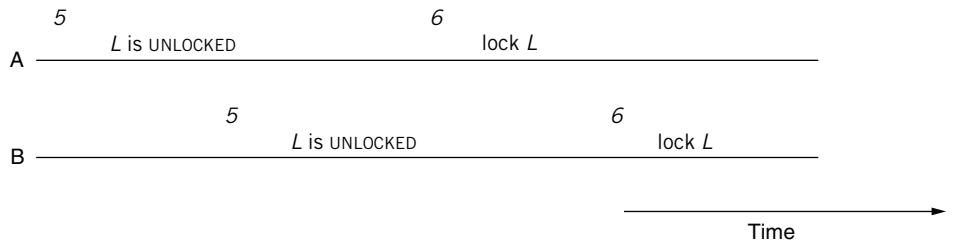
A problem related to deadlock is *livelock*—an interaction among a group of threads in which each thread is repeatedly performing some operations but is never able to complete the whole sequence of operations. An example of livelock is given in Sidebar 5.2, which presents an algorithm to implement ACQUIRE and RELEASE.

5.2.6 Implementing ACQUIRE and RELEASE

A correct implementation of ACQUIRE and RELEASE must enforce the single-acquire protocol. Several threads may attempt to acquire the lock at the same time, but only one should succeed. This requirement makes the implementation of locks challenging. In essence, we must make sure that ACQUIRE itself is a before-or-after action.

To see what goes wrong if ACQUIRE is not a before-or-after action, consider the too-simple implementation of ACQUIRE as shown in Figure 5.7. This implementation is broken because it has a race condition. If two threads labeled A and B call

FAULTY_ACQUIRE at the same time, the threads may execute the statements in the order A5, B5, A6, B6, which corresponds to the following timing diagram:



The result of this sequence of events is that both threads acquire the lock, which violates the single-acquire protocol.

The faulty ACQUIRE has a multistep operation on a shared variable (the lock), and we must ensure in some way that ACQUIRE itself is a before-or-after action. Once ACQUIRE is a before-or-after action, we can use it to turn arbitrary multistep operations on shared variables into before-or-after actions. This reduction is an example of a technique called *bootstrapping*, which resembles an inductive proof. Bootstrapping means that we look for a systematic way to reduce a general problem (e.g., making multistep operations on shared variables before-or-after actions) to some much-narrower particular version of the same problem (e.g., making an operation on a single shared lock a before-or-after action). We then solve the narrow problem using some specialized method that might work for only that case because it takes advantage of the specific situation. The general solution then consists of two parts: a method for solving the special case and a method for reducing the general problem to the special case. In the case of ACQUIRE, the solution for the specific problem is either building a special hardware instruction that is itself a before-or-after action or programming very carefully.

```
1    structure lock
2        integer state
3
4    procedure FAULTY_ACQUIRE (lock reference L)
5        while L.state = LOCKED do nothing      // spin until L is UNLOCKED
6        L.state ← LOCKED                       // the while test failed, got the lock
7
8    procedure RELEASE (lock reference L)
9        L.state ← UNLOCKED
```

FIGURE 5.7

Incorrect implementation of ACQUIRE. LOCKED and UNLOCKED are constants that have different values; for example, LOCKED is 1 and UNLOCKED is 0.

We first look at a solution involving a special instruction, *Read and Set Memory* (RSM). RSM performs the statements in the block **do atomic** as a before-or-after action:

```
1    procedure RSM (reference mem)     // RSM memory location mem
2        do atomic
3            r ← mem                   // Load value stored at mem into r
4            mem ← LOCKED              // Store LOCKED into memory location mem
5        return r
```

Most modern computers implement some version of the RSM procedure in hardware, as an extension to the memory abstraction. RSM is then often called *test-and-set*; see Sidebar 5.1. For the RSM instruction to be a before-or-after action, the bus arbiter that controls the bus connecting the processors to the memory must guarantee that the LOAD (line *3*) and STORE (line *4*) instruction execute as before-or-after actions—for example, by allowing the processor to read a value from a memory location and to write a new value into that same location in a single bus cycle. We have thus pushed the problem of providing a before-or-after action down to the bus arbiter, a piece of hardware whose precise function is turning bus operations into before-or-after

Sidebar 5.1 RSM, Test-and-Set, and Avoiding Locks RSM is often called "test-and-set" or "test-and-set-locked" for accidental reasons. An early version of the instruction tested the lock and performed the store only if the test showed that the lock was not set. The instruction also set a bit that the software could test to find out whether or not the lock had been set. Using this instruction, one can implement the body of ACQUIRE as follows:

while TEST_AND_SET (*L*) = LOCKED **do nothing**

This version appears to be shorter than the one shown in Figure 5.8, but the hardware performs a test that is redundant. Thus, later hardware designers removed the test from test-and-set, but the name stuck.

In addition to RSM, there are many other instructions, including "test-and-test-and-set" (which allows for a more efficient implementation of a spin lock) and COMPARE_AND_SWAP (*v1, m, v2*) (which compares, in a before-or-after action the content of a memory location *m* to the value *v1* and, if they are the same, stores *v2* in *m*). The "compare-and-swap" instruction can be used, for example, to implement a linked list in which threads can insert elements concurrently without having to use locks, avoiding the risk of spinning until other threads have completed their insert [see Suggestions for Further Reading 5.5.8 and 5.5.9]. Such implementations are called *non-blocking*.

The Linux kernel uses yet another form of coordination that avoids locks. It is called read-copy update and is tailored to data structures that are mostly read and infrequently updated [see Suggestions for Further Reading 5.5.7].

```
1     procedure ACQUIRE (lock reference L)
2         R1 ← RSM (L.state)                  // read and set lock L
3         while R1 = LOCKED do                // was it already locked?
4             R1 ← RSM(L.state)               // yes, do it again, till we see it wasn't
5
6     procedure RELEASE (lock reference L)
7         L.state ← UNLOCKED
```

FIGURE 5.8

ACQUIRE and RELEASE using RSM.

actions: the arbiter guarantees that if two requests arrive at the same time, one of those requests is executed completely before the other begins.

Using the RSM instruction, we can implement any other before-or-after action. It is the one essential before-or-after action from which we can bootstrap any other set of before-or-after actions. Using RSM, we can implement ACQUIRE and RELEASE as shown in Figure 5.8. This implementation follows the single-acquire protocol: if *L* is LOCKED, then one thread has the lock *L*; if *L* contains UNLOCKED, then no thread has acquired the lock *L*.

To see that the implementation is correct, let's assume that *L* is UNLOCKED. If some thread calls ACQUIRE (*L*), then after RSM, *L* is LOCKED and R1 contains UNLOCKED, so that thread has acquired the lock. The next thread that calls ACQUIRE (*L*) sees LOCKED in R1 after the RSM instruction, signaling that some other thread holds the lock. The thread that tried to acquire will spin until R1 contains UNLOCKED. When releasing a lock, no test is needed, so an ordinary STORE instruction can do the job without creating a race condition.

This implementation assumes that the shared memory provides read/write coherence. For example, if a manager thread sets *L* to UNLOCKED on line 7, then we assume that the thread observes that store and falls out of the spinning loop on line *3* in ACQUIRE. Some memories provide more relaxed semantics than read/write coherence; in that case, additional mechanisms are needed to make this program work correctly.

With this implementation, even a single thread can deadlock itself by calling ACQUIRE twice on the same lock. With the first call to ACQUIRE, the thread obtains the lock. With the second call to ACQUIRE the thread deadlocks, since some thread (itself) already holds the lock. By storing the thread identifier of the lock's owner in *L* (instead of true or false), ACQUIRE could check for this problem and return an error.

Problem set *6* explores concurrency issues using a SET-AND-GET remote procedure call, which executes as a before-or-after action.

5.2.7 Implementing a Before-or-After Action Using the One-Writer Principle

The RSM instruction can also be implemented without extending the memory abstraction. In fact, one can implement RSM as a procedure in software using ordinary load and store instructions, but such implementations are complex. The key

Sidebar 5.2 Constructing a Before-or-After Action Without Special Instructions

In 1959, E. Dijkstra, a well-known Dutch programmer and researcher, posed to his colleagues the problem of providing a before-or-after action with ordinary read and write instructions as an amusing puzzle. Th. J. Dekker provided a solution for two threads, and Dijkstra generalized the idea into a solution for an arbitrary number of threads [Suggestions for Further Reading 5.5.2]. Subsequently, numerous researchers have looked for provable, efficient solutions. We present a simple implementation of RSM based on L. Lamport's solution. Lamport's solution, like other solutions, relies on the existence of a bus arbiter that guarantees that any single LOAD or STORE is a before-or-after action with respect to every other LOAD and STORE. Given this assumption, RSM can be implemented as follows:

```
shared boolean flag[N]                          // one boolean per thread

1   procedure RSM (lock reference L)            // set lock L and return old value
2       do forever                              // me is my index in flag
3           flag[me] ← TRUE                     // warn other threads
4           if ANYONE_ELSE_INTERESTED (me) then // is another thread warning us?
5            flag[me] ← FALSE                   // yes, reset my warning, try again
6           else
7            R ← L.state                        // set R to value of lock
8            L.state ← LOCKED                   // and set the lock
9            flag[me] ← FALSE
10           return R
11
12  procedure ANYONE_ELSE_INTERESTED (me)       // is another thread updating L?
13      for i from 0 to N–1 do
14          if i ≠ me and flag[i] = TRUE then return TRUE
15      return FALSE
```

To guarantee that RSM is indeed a before-or-after action, we need to assume that each entry of the shared array is in its own memory cell, that the memory provides read/write coherence for memory cells, and that the instructions execute in program order, as we did for the sender and receiver in Figure 5.5.

Under these assumptions, RSM ensures that the shared variable *L* is never written by two threads at the same time. Each thread has a unique number, *me*. Before *me* is allowed to write *L*, it must express its interest in writing *L* by setting *me*'s entry in the boolean array *flag* (line *3*) and check that no other thread is interested in writing *L* (line *4*). If no other thread has expressed interest, then *me* acquires *L* (line *8*).

If two threads A and B call RSM at the same time, either A or B may acquire *L*, or both may retry, depending on how the shared memory system orders the accesses of A and B to the *flag*[i] array. There are three cases:

1. A sets *flag*[A], calls ANYONE_ELSE_INTERESTED, and reads flags at least as far as *flag*[B] before B sets *flag*[B]. In this case, A sees no other flags set and proceeds to acquire *L*;

(*Sidebar continues*)

B discovers A's flag and tries again. On its next try, B encounters no flags, but by the time B writes LOCKED to *L*, *L* is already set to LOCKED, so B's write will have no effect.

2. B sets *flag*[B], calls ANYONE_ELSE_INTERESTED, and reads flags at least as far as *flag*[A] before A sets *flag*[A]. In this case, B sees no other flags set and proceeds to acquire *L*; A discovers B's flag and tries again. On its next try, A encounters no flags, but by the time A writes *locked* to *L*, *L* is already set to *locked*, so A's write will have no effect.
3. A sets *flag*[A] and B sets *flag*[B] before either of them gets far enough through ANYONE_ELSE_INTERESTED to reach the other's flag location. In this case, both A and B reset their own *flag*[*i*] entries and try again. On the retry, all three cases are again possible.

The implementation of RSM has a livelock problem because the two threads A and B might end up in the final case (neither of them gets to update *L*), every time they retry. RSM could reduce the chance of livelock by inserting a random delay before retrying, a technique called *random backoff*. Chapter 7 [on-line] will refine the random backoff idea to make it applicable to a wider range of problems.

This implementation of RSM is not the most efficient one; it is linear in the number of threads because ANYONE_ELSE_INTERESTED reads all but one element of the array *flag*. More efficient versions of RSM exist, but even the best implementation [Suggestions for Further Reading 5.5.3] requires two loads and five stores (if there is no contention for *L*), which can be proven to be optimal under the given assumptions.

problem that our implementation without RSM of ACQUIRE has is that several threads are attempting to *modify* the same shared variable (*L* in our example). For two threads to read *L* concurrently is fine (the bus arbiter ensures that LOADs are before-or-after actions, and both threads will read the same value), but reading *and* modifying *L* is a multistep operation that must be performed as a before-or-after action. If not, this multistep operation can lead to a race condition in which the outcome may be a violation of the single-acquire protocol. This observation suggests an approach to implementing RSM based on the one-writer principle: ensure that only *one* thread modifies *L*.

Sidebar 5.2 describes a software solution that follows that approach. This software solution is complex compared to the hardware implementation of RSM. To ensure that only one thread writes *L*, the software solution requires an array with one entry per thread. Such an array must be allocated for each lock. Moreover, the number of memory accesses to acquire a lock is linear in the number of threads. Also, if threads are created dynamically, the software solution requires a more complex data structure than an array. Between the need for efficiency and the requirement for an array of unpredictable size, designers generally implement RSM as a hardware instruction that invokes a special bus cycle.

If one follows the *one-writer principle* carefully, one can write programs without locks (for example, as in Figure 5.5). This approach without locks can improve a program's performance because the expense of locks is avoided, but eliminating locks makes it more difficult to reason about the correctness.

The designers of the computer system for the space shuttle used many threads sharing many variables, and they deployed a systematic design approach to encourage a correct implementation. Designed in the late 1970s and early 1980s, the computers of the space shuttle were not efficient enough to follow the principled way of protecting all shared variables using locks. Understanding the risks of sharing variables among concurrent threads, however, the designers followed a rule that the program declaration for each unprotected shared variable must be accompanied by a comment, known as an *alibi*, explaining why no race conditions can occur even though that variable is unprotected. At each new release of the software, a team of engineers inspects all alibis and checks whether they still hold. Although this method has a high verification overhead, it helps discover many race conditions that otherwise might go undetected until too late. The use of alibis is an example of design for iteration.

5.2.8 Coordination between Synchronous Islands with Asynchronous Connections

As has been seen in this chapter, all implementations of before-or-after actions rely on bootstrapping from a properly functioning hardware arbiter. This reliance should catch the attention of hardware designers, who are aware that under certain conditions, it can be problematic (indeed, theoretically impossible) to implement a perfect arbiter. This section explains why and how hardware designers deal with this problem in practice. System designers need to be aware of how arbiters can fail, so that they know what questions to ask the designer of the hardware on which they rely.

The problem arises at the interface between asynchronous and synchronous components, when an arbiter that provides input to a synchronous subsystem is asked to choose between two asynchronous but closely spaced input signals. An asynchronous-input arbiter can enter a metastable state, with an output value somewhere between its two correct values or possibly oscillating between them at a high rate.* After applying asynchronous signals to an arbiter, one must therefore wait for the arbiter's output to settle. Although the probability that the output of the arbiter has not settled falls exponentially fast, for any given delay time some chance always remains that the arbiter has not settled yet, and a sample of its output may find it still changing. By waiting longer, one can reduce the probability of it not having settled to as small a figure as necessary for any particular application, but it is impossible to drive it to zero within a fixed time. Thus if the component that acquires the output of the arbiter is synchronous, when its clock ticks there is a chance that the component's

*Our colleague Andreas Reuter points out that the possibility that an arbiter may enter a metastable state has been of concern since antiquity: "How long halt ye between two opinions"?—*1 Kings* 18:21.

input (that is, the arbiter's output) is not ready. When that happens, the component may behave unpredictably, launching a chain of failure. Although the arbiter itself will certainly come to a decision at some point, not doing so before the clock ticks is known as *arbiter failure*.

Arbiter failure can be avoided in several ways:

- Synchronize the clocks of the two components. If the two processors, the arbiter, and the memory all operate with a common clock (more precisely, all of their interfaces are synchronous), arbiter design becomes straightforward. This technique is used, for example, to arbitrate access within some chips that have several processors.
- Design arbiters with multiple stages. Multiple stages do not eliminate the possibility of arbiter failure, but each additional stage multiplicatively reduces the probability of failure. The strategy is to provide enough stages that the probability of failure is so low that it can be neglected. With current technology, two or three stages are usually sufficient, and this technique is used in most interfaces between asynchronous and synchronous devices.
- Stop the clock of the synchronous component (thus effectively making it asynchronous) and wait for the arbiter's output to settle before restarting. In modern high-performance systems, clock distribution requires continuous ticks to provide signals for correcting phase errors, so one does not often encounter this technique in practice.
- Make all components asynchronous. The component that takes the output of the arbiter then simply waits until the arbiter reports that it has settled. A flurry of interest in asynchronous circuit design arose in the 1970s, but synchronous circuits proved to be easier to design and so won out. However, as clock speeds increase to the point that it is difficult to distribute clock even across a single chip, interest is reawakening.

Communication across a network link is nearly always asynchronous, communication between devices in the same box (for example, between a disk drive and a processor) is usually asynchronous, and as mentioned in the last bullet above, as advancing technology reduces gate delays, it is becoming challenging to maintain a common, fast-enough clock all the way across even a single chip. Thus, within-chip intercommunication is becoming more network-like, with synchronous islands connected by asynchronous links (see, for example, Suggestions for Further Reading 1.6.3).

As pointed out, arbiter failure is an issue only at the boundary between synchronous and asynchronous components. Over the years, that boundary has moved with changing technology. The authors are not aware of any current implementations of RSM ( ) or their equivalents that cross a synchronous/asynchronous boundary (in other words, current multiprocessor practice is to use the method of the first bullet above). Thus, before-or-after atomicity based on RSM ( ) is not at risk of arbiter failure. But that was not true in the past, and it may not be true again at

some point in the future. The system designer thus needs to be aware of where arbiters are being used and verify that they are specified appropriately for the application.

5.3 ENFORCING MODULARITY IN MEMORY

The implementation of bounded buffers took advantage of the fact that all threads share the same physical memory (see Figure 5.4 on page 209), but sharing memory does not enforce modularity well. A program may calculate a shared address incorrectly and write to a memory location that logically belongs to another module. To enforce modularity, we must ensure that the threads of one module cannot overwrite the data of another module. This section introduces domains and a memory manager to enforce memory boundaries, assuming that the address space is very large (i.e., so large that we can consider it unlimited). In Section 5.4, we will remove that assumption.

5.3.1 Enforcing Modularity with Domains

To contain the memory references of a thread, we restrict the thread's references to a *domain*, a contiguous range of memory addresses. When a programmer calls `ALLOCATE_THREAD`, the programmer specifies a domain in which the thread is to run. The thread manager records a thread's domain.

To enforce the rule that a thread should refer only to memory within its domain, we add a domain register to each processor and introduce a special interpreter, a *memory manager*, that is typically implemented in hardware and placed between a processor and the bus (see Figure 5.9). A processor's domain register contains the

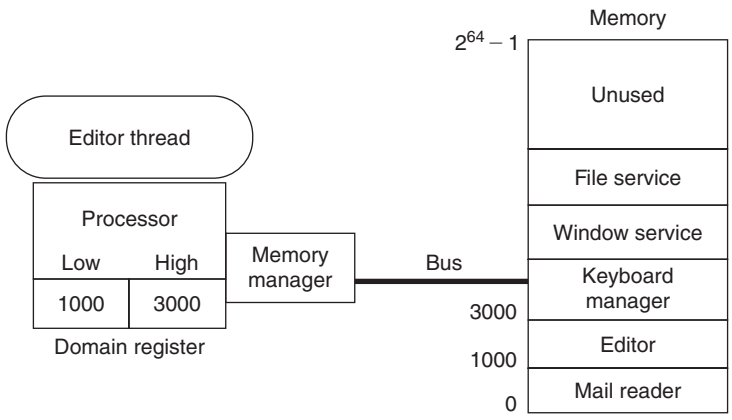


FIGURE 5.9

An editor thread running with its domain.

lowest (*low*) and highest address (*high*) that the currently running thread is allowed to use. ALLOCATE_THREAD loads the processor's domain register with the thread's domain.

The memory manager checks for each memory reference that the address is equal or higher than *low* and smaller than *high*. If it is, the memory manager issues the corresponding bus request. If not, it interrupts the processor, signaling a *memory reference exception*. The exception handler can then decide what to do. One option is to deliver an error message and destroy the thread.

This design ensures that a thread can make references only to addresses that are in its domain. Threads cannot overwrite or jump to memory locations of other threads.

This domain design achieves the main goal, but it lacks a number of desirable features:

1. A thread may need more than one domain. By using many domains, threads can control what memory they share and what memory they keep private. For example, a thread might allocate a domain for a bounded buffer and share that domain with another thread, but allocate private domains for the text of its program and private data structures.

2. A thread should be unable to change its own domain. That is, we must ensure that the thread cannot change the content of its processor's domain register directly or indirectly. If a thread can change the content of its processor's domain register, then the thread can make references to addresses that it shouldn't.

The rest of Section 5.3 adds these features.

5.3.2 Controlled Sharing Using Several Domains

To allow for sharing, we extend the design to allow each thread to have several domains and give each processor several domain registers, for the moment, as many as a thread needs. Now a designer can partition the memory of the programs shown in Figure 5.9 and control sharing. For example, a designer may split a client into four separate domains (see Figure 5.10): one domain containing the program text for the client thread, one domain containing the data for the client, one domain containing the stack of the client thread, and one domain containing the bounded message buffer. The designer may split a service in the same way. This setup allows both threads to use the shared bounded buffer domain, but restricts the other references of the threads to their private domains.

To manage this hardware design, we introduce a software component to the memory manager, which provides the following interface:

- *base_address* ← ALLOCATE_DOMAIN (*size*): Allocate a new domain of *size* bytes and return the base address of the domain.

- MAP_DOMAIN (*base_address*): Add the domain starting at address *base_address* to the calling thread's domains.

The memory manager can implement this interface by keeping a list of memory regions that are not in use, allocate *size* bytes of memory on an ALLOCATE_DOMAIN request, and maintain a *domain table* with allocated domains. An entry in the domain table records the *base_address* and *size*.

MAP_DOMAIN loads the domain's bounds from the domain table into a domain register of the thread's processor. If two or more threads map a domain, then that domain is shared among those threads.

We can improve the control mechanism by extending each domain register to record access permissions. In a typical design, a domain register might include three bits, which separately control permission to READ, WRITE, or EXECUTE (i.e., retrieve and use as an instruction) any of the bytes in the associated domain.

With these permissions, the designer can give the threads in Figure 5.10 EXECUTE and READ permissions to their text domains, and READ and WRITE permissions to their stack domains and the shared bounded buffer domain. This setup prevents a thread from taking instructions from its stack, which can help catch programming mistakes and also help avoid buffer overrun attacks (see Sidebar 11.4 [on-line]). Giving the program text only READ and EXECUTE permissions helps avoid the mistake of accidentally writing data into the text of the program.

The permissions also allow more controlled sharing: one thread can have access to a shared domain with only READ permission, whereas another thread can have READ and WRITE permissions.

To provide permissions, we modify the MAP_DOMAIN call as follows:

- MAP_DOMAIN (*base_address, permission*): loads the domain's bounds from the domain table into one of the calling thread's domain registers with permission *permission*.

To check permissions, the memory manager must know which permissions are needed for each memory reference. A LOAD instruction requires READ permission for its address, and thus the memory manager must check that the address is in a domain with READ access. A STORE instruction requires WRITE permission for its address, and

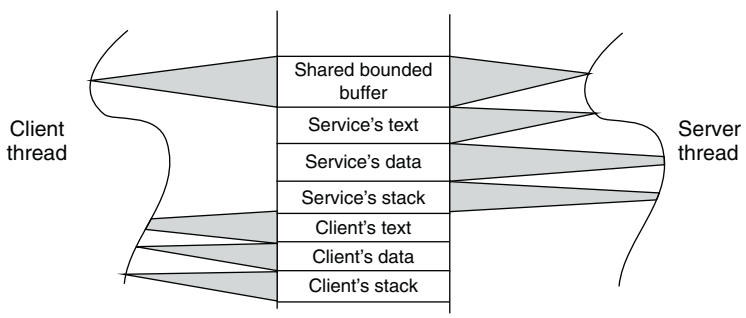


FIGURE 5.10

A client and service, each with three private domains and one shared domain.

thus the memory manager must check that the address is in a domain with WRITE access. To execute an instruction at the address in the PC requires EXECUTE permission. The domain holding instructions may also require READ permission because the program may have stored constants in the program text.

The pseudocode in Figure 5.11 details the check performed by the memory manager. Although we describe the function of the memory manager using pseudocode, in practice the memory manager is a hardware device that implements its function in digital circuitry. In addition, the memory manager is typically integrated with the processor so that the address checks run at the speed of the processor. As Section 5.3 develops, we will add more functions to the memory manager, some of which may be implemented in software as part of the operating system. Later, in Section 5.4.4, we discuss the trade-offs involved in implementing parts of the memory manager in software.

As shown in the figure, on a memory reference, the memory manager checks all the processor's domain registers. For each domain register, the memory manager calls CHECK_DOMAIN, which takes three arguments: the address the processor requested, a bit mask with the permissions needed by the current instruction, and the domain register. If *address* falls between *low* and *high* of the domain and if the permissions needed are a subset of permissions authorized for the domain, then CHECK_DOMAIN returns TRUE and the memory manager will issue the desired bus request. If *address* falls between *low* and *high* of the domain but the permissions needed aren't sufficient, then CHECK_DOMAIN interrupts the processor, indicating a memory reference exception as before. Now, however, it is useful to demultiplex the memory reference exception in two different categories: *illegal memory reference exception* and *permission error exception*. If *address* doesn't fall in any domain, the exception handler indicates an illegal memory reference exception. If the address falls in a domain, but the threads didn't have sufficient permissions, the exception handler indicates a permission error exception.

```
1   procedure LOOKUP_AND_CHECK (integer address, perm_needed)
2       for each domain do                  // for each domain register of this processor
3           if CHECK_DOMAIN (address, perm_needed, domain) return domain
4       signal memory_exception
5
6   procedure CHECK_DOMAIN (integer address, perm_needed, domain) returns boolean
7       if domain.low ≤ address and address < domain.high then
8           if PERMITTED (perm_needed, domain.permission) then return TRUE
9           else signal permission_exception
10      return FALSE
11
12  procedure PERMITTED (perm_needed, perm_authorized) returns boolean
13      if perm_needed ⊂ perm_authorized then return TRUE // is perm_needed a subset?
14      else return FALSE               // permission violation
```

FIGURE 5.11

The memory manager's pseudocode for looking up an address and checking permissions.

The demultiplexing of memory reference exceptions can be implemented either in hardware or software. If implemented in hardware, the memory manager signals an illegal memory reference exception or a permission error exception to the processor. If implemented in software, the memory manager signals a memory reference exception, and the exception handler for memory reference exceptions demultiplexes it by calling either the illegal memory reference exception handler or the permission error exception handler. As we will see in Chapter 6 (see Section 6.2.2), we will want to refine the categories of memory exceptions further. In processors in the field, some of the demultiplexing is implemented in hardware with further demultiplexing implemented in software in the exception handler.

In practice, only a subset of the possible combinations of permissions are useful. The ones used are: READ permission, READ and WRITE permissions, READ and EXECUTE permissions, and READ, WRITE, and EXECUTE permissions. The READ, WRITE, and EXECUTE combination of permissions might be used for a domain that contains a program that generates instructions and then jumps to the generated instructions, so-called self-modifying programs. Supporting self-modifying programs is risky, however, because this also allows an adversary to write a new procedure as data into a domain (e.g., using a buffer overrun attack) and then execute that procedure. In practice, self-modifying programs have proven to be more trouble than they are worth, except to system crackers. A domain with only EXECUTE permission or with just WRITE and EXECUTE permissions isn't useful in practice.

If memory-mapped I/O (described in Section 2.3.1) is used, then domain registers can also control which devices a thread can use. For example, a keyboard manager thread may have access to a domain that corresponds to the registers of the keyboard controller. If none of the other threads has access to this domain, then only the keyboard manager thread has access to the keyboard device. Thus, the same technique that controls separation of memory ranges can also control access to devices.

The memory manager can implement security policies because it controls which threads have access to which parts of memory. It can deny or grant a thread's request for allocating a new domain or for sharing an existing domain. In the same way, it can control which threads have access to which devices. How to implement such security policies is the topic of Chapter 11 [on-line].

5.3.3 More Enforced Modularity with Kernel and User Mode

Domain registers restrict the addresses to which a thread can make reference, but we must also ensure that a thread cannot change its domains. That is, we need a mechanism to control changes to the *low*, *high*, and *permission* fields of a domain register. To complete the enforcement of domains, we modify the processor to prevent threads from overwriting the content of domain registers as follows:

- Add one bit to the processor indicating whether the processor is in ***kernel mode*** or ***user mode***, and modify the processor as follows. Instructions can change the value of the processor's domain registers only in kernel mode, and instructions

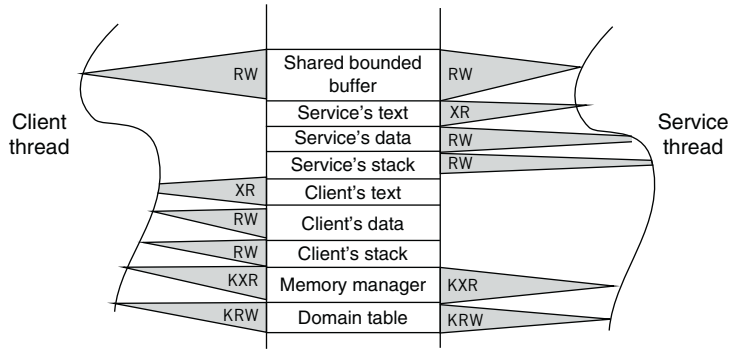


FIGURE 5.12

Threads with a kernel domain containing the memory manager and its domain table.

that attempt to change the domain register in user mode generate an *illegal instruction exception*. Similarly, instructions can change the mode bit only in kernel mode.

- Extend the set of permissions for a domain to include KERNEL-ONLY, and modify the processor to make it illegal for threads in user mode to reference addresses in a domain with KERNEL-ONLY permission. A thread in user mode that attempts to read or write memory with KERNEL-ONLY permission causes a permission error exception.
- Switch to kernel mode on an interrupt and on an exception so that the handler can process the interrupt (or exception) and invoke privileged instructions.

We can use the mechanisms as illustrated in Figure 5.12. Compared to Figure 5.10, each thread has two additional domains, which are marked K for KERNEL-ONLY. A thread must be in kernel mode to be able to make references to this domain. This domain contains the program text and data for the memory manager. These mechanisms ensure that a thread running in user mode cannot change its processor's domain registers; only when a thread executes in kernel mode can it change the processor domain registers. Furthermore, because the memory manager and its table are in kernel domains, a thread in user mode cannot change its domain information. We see that the kernel/user mode bit helps in enforcing modularity by restricting what threads in user mode can do.

5.3.4 Gates and Changing Modes

Because threads running in user mode cannot invoke procedures of the memory manager directly, a thread must have a way of changing from user mode to kernel mode and entering the memory manager in a controlled manner. If a thread could enter at an arbitrary address, it might create problems; for example, if a thread could enter a

domain with kernel permission at the instruction that sets the user-mode bit to KERNEL, it might be able to gain control of the processor with kernel privileges. To avoid this problem, a thread may enter a kernel domain only at certain addresses, called *gates*.

We implement gates by adding one more special instruction to the processor, the *supervisor call instruction* (SVC), which specifies in a register the name for the intended gate. Upon executing the SVC instruction, the processor performs two operations as one action:

1. Change the processor mode from user to kernel.
2. Set the PC to an address predefined by the hardware, the entry point of the gate manager.

The gate manager now has control in kernel mode; it can call the appropriate procedures to serve the thread's request. Typically, gate names are numbers, and the gate manager has a table that records for each gate number the corresponding procedure. For example, the table might map gate 0 to ALLOCATE_DOMAIN, gate 1 to MAP_DOMAIN, gate 2 to ALLOCATE_THREAD, and so on.

Implementing SVC has a slight complication: the steps to enter the kernel must happen as a before-or-after action: they must all be executed *without* interruption. If the processor can be interrupted in the middle of these steps, a thread might end up in kernel mode but with the program counter still pointing to an address in one of its user-level domains. Now the thread is executing instructions from an application module in kernel mode. To avoid this potential problem, processors complete all steps of an SVC instructions before executing another instruction.

When the thread wants to return to user mode, it executes the following instructions:

1. Change mode from kernel to user.
2. Load the program counter from the top of the stack into the processor's PC.

Processors don't have to perform these steps as a before-or-after action. After step 1, it is fine for a processor to return to kernel mode, for example, to process an outstanding interrupt.

The return sequence assumes that a thread has pushed the return address on its stack before invoking SVC. If the thread hasn't done so, then the worst that can happen when the thread returns to user mode is that it resumes at some arbitrary address, which might cause the thread to fail (as with many other programming errors), but it cannot create a problem for a domain with KERNEL-ONLY permission because the thread cannot refer to that domain in user mode.

The difference between entering and leaving kernel mode is that on leaving, the value loaded in the program counter isn't a predefined value. Instead, the kernel sets it to the saved address.

Gates can also be used to handle interrupts and exceptions. If the processor encounters an interrupt, the processor enters a special gate for interrupts and the gate manager dispatches the interrupt to the appropriate handler, based on the source of the interrupt (clock interrupt, permission error, illegal memory reference, divide by zero, etc.). Some

processors have a different gate for errors caused by exceptions (e.g., a permission error); others have one gate for interrupts (e.g., clock interrupt) and exceptions.

Problem set *9* explores in a minimal operating system the interactions between hardware and software for setting modes and handling interrupts.

5.3.5 Enforcing Modularity for Bounded Buffers

The implementation of SEND and RECEIVE in Figure 5.6 assumes that the sending and receiving threads share the bounded buffer, using, for example, a shared domain, as shown in Figure 5.12. This setup enforces a boundary between all domains of the threads, except for the domain containing the shared buffer. A thread can modify the shared buffer accidentally because both threads have write permissions to the shared domain. Thus, an error in one thread could indirectly affect the other thread; we would like to avoid that and enforce modularity for the bounded buffer.

We can protect the shared bounded buffer, too, by putting the buffer in a shared kernel domain (see Figure 5.13). Now the threads cannot directly write the shared buffer in user mode. The threads must transition to kernel mode to copy messages into the shared buffer. In this design, SEND and RECEIVE are supervisor calls. When a thread invokes SEND, it changes to kernel mode and copies the message from the sending thread's domain into the shared buffer. When the receiving thread invokes RECEIVE, it changes to kernel mode and copies a message from the shared buffer into the receiving's domain. As long as the program that is running in kernel mode is written carefully, this design provides stronger enforced modularity because threads in user mode have no direct access to a bounded buffer's messages.

This stronger enforced modularity comes at a performance cost for performing supervisor calls for SEND and RECEIVE. This cost can be significant because transitions between user mode and kernel mode can be expensive. The reason is that a processor typically maintains state in its pipeline and a cache as a speedup mechanism. This state

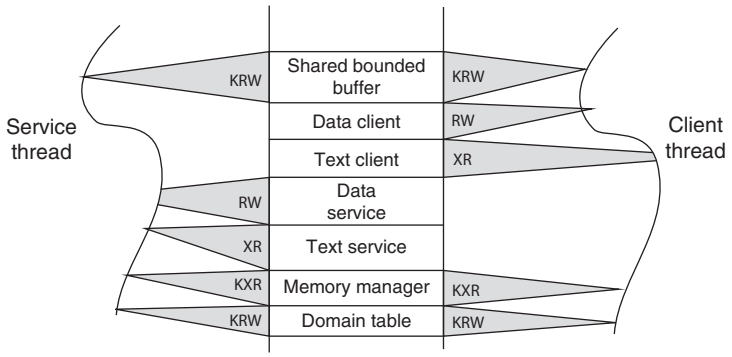


FIGURE 5.13

Threads with a kernel domain containing the shared buffer, the memory manager, and the domain table.

may have to be flushed or invalidated on a user-kernel mode transition because otherwise the processor may execute instructions that are still in the pipeline incorrectly.

Researchers have come up with techniques to reduce the performance cost by optimizing the kernel code paths for the SEND and RECEIVE supervisor calls, by having a combined call that sends and receives, by cleverly setting up domains to avoid the cost of copying large messages, by passing small arguments through processor registers, by choosing a suitable layout of data structures that reduces the cost of user-kernel mode transitions, and so on [Suggestions for Further Reading 6.2.1, 6.2.2, and 6.2.3]. Problem set 7 illustrates a lightweight remote procedure call implementation.

5.3.6 The Kernel

The collection of modules running in kernel mode is usually called the kernel program, or *kernel* for short. A question that arises is how the kernel and the first domain come into existence. Sidebar 5.3 details how a processor starts in kernel mode with domain checking disabled, how the kernel can then bootstrap the first domain, and how the kernel can create user-level domains.

The kernel is a trusted intermediary because it is the only program that can execute privileged instructions (such as storing to a processor's domain registers) and the application modules rely on the kernel to operate correctly. Because the kernel must be a trusted intermediary for the memory manager hardware, many designers also make the kernel the trusted intermediary for all other shared devices, such as the clock, display, and disk. Modules that manage these devices must then be part of the kernel program. In this design, the window manager module, network manager module, and file manager module run in kernel mode. This kernel design, in which most of the operating system runs in kernel mode, is called a *monolithic kernel* (see Figure 5.14).

In kernel mode, errors such as dividing by zero are fatal and halt the computer because these errors are typically caused by programming mistakes in the kernel program, from which there is no easy way to recover. Since kernel errors are fatal, we must program and structure the kernel carefully.

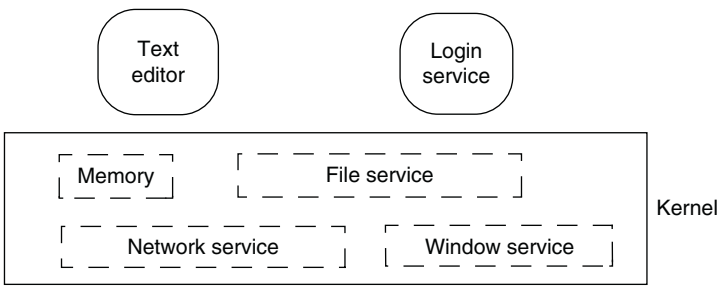


FIGURE 5.14

Monolithic organization: the kernel implements the operating system.

Sidebar 5.3 Bootstrapping an Operating System When the user switches on the power for the computer, the processor starts with all registers set to zero; thus, the user-mode bit is off. The first instruction the processor executes is the instruction at address 0 (the value of the pc register). Thus after a reset, the processor fetches its first instruction from address 0.

Address 0 typically corresponds to a read-only memory (ROM). This memory contains some initial code, the boot code, a rudimentary kernel program, which loads the full kernel program from a magnetic disk. The computer manufacturer burns into the read-only memory the boot program, after which the boot program cannot be changed. The boot program includes a rudimentary file system, which finds the kernel program (probably written by a software manufacturer) at a pre-agreed location on disk. The boot code reads the kernel into physical memory and jumps to the first instruction of the kernel.

Bootstrapping the kernel through a small boot program provides modularity. The hardware and software manufacturers can develop their products independently, and users can change kernels, for example, to upgrade to a newer version or to use a different kernel vendor, without having to modify their hardware.

Sometimes there are multiple layers of booting to handle additional constraints. For example, the first boot loader may be able to load only a single block, which can be too small to hold the rudimentary kernel program. In such cases, the boot code may load first an even simpler kernel program, which then loads the rudimentary kernel program, which then loads the kernel program.

Once it is running, the kernel allocates a thread for itself. This thread allocation involves allocating a domain for use as a stack so that it can invoke procedure calls, allowing the rest of the kernel to be written in a high-level language. It may also allocate a few other domains, for example, one for the domain table.

Once the kernel has initialized, it typically creates one or more threads to run non-kernel services. It allocates to each service one or more domains (e.g., one for program text, one for a stack, and one for data). The kernel typically preloads some of the domains with the program text and data of the non-kernel services. A common solution to locating the program text and data is to assume that the first non-kernel program, like the kernel program, is at a predefined address on the magnetic disk or part of the data of the kernel program.

Once thread is running in user mode, it can reenter the kernel using a gate for a kernel procedure. Using the kernel procedures, the user-level thread can create more threads, allocate domains for these threads, and, when done, exit.

Errors by threads in user mode (e.g., dividing by zero or using an address that is not in the thread's domains or violates permissions) cause an exception, which changes the processor to kernel mode. The exception handler can then clean up the thread.

We would like to keep the kernel small because the number of bugs in a program is at least proportional to the size of a program—and some even argue to the square of the size of program. In a monolithic kernel, if the programmer of the file manager module has made an error, the file manager module may overwrite kernel data structures unrelated to the file system, thus causing unrelated parts of the kernel to fail.

The *microkernel* architecture structures the operating system itself in a client/service style (see Figure 5.15). By applying the idea of enforced modularity to the operating system itself, we can avoid some of the major problems of a monolithic organization. In the microkernel architecture, system modules run in user mode in their own domain, as opposed to being part of a monolithic kernel. The microkernel itself implements a minimal set of abstractions, primarily domains to contain modules, threads to run programs, and virtual communication links to allow modules to send messages to one another. The kernel described in this chapter with its interface shown in Table 5.1 is an example of a microkernel.

In the microkernel organization, for example, the window service module runs in its own domain with access to the display, the file service module runs in its own domain with access to a disk extent, and the database service runs in its own domain with its own disk extent. Clients of the services communicate with them by invoking remote procedure calls, whose stubs in turn invoke the SEND and RECEIVE supervisor calls. An early, clean design for a microkernel is presented by Hansen [Suggestions for Further Reading 5.1.1].

A benefit of the microkernel organization is that errors are contained within a module, simplifying debugging. A programming error in the file service module affects only the file service module; no other module ever has its internal data structures unintentionally modified because of an error by the programmer of the file service module. If the file service fails, a programmer of the file service can focus on debugging the file service and rule out the other services immediately. In contrast with the monolithic kernel approach, it is difficult to attribute an error in the kernel to a particular module because the modules aren't isolated from each other and an error in one module may be caused by a flaw in another module.

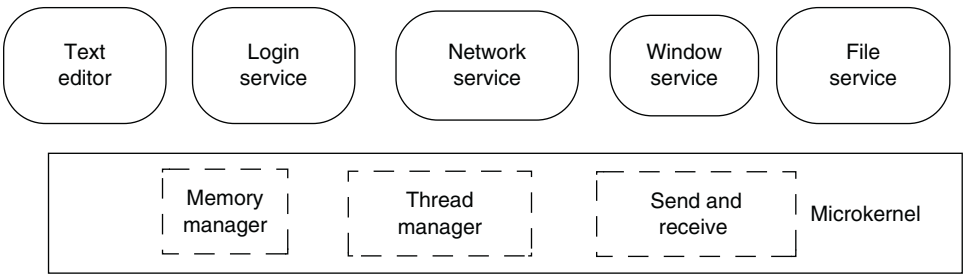


FIGURE 5.15

Microkernel organization: the operating system organized using the client/service model.

In addition, if the file service fails, the database service may be able to continue operating. Of course, if the file service module fails, its clients cannot operate, but they may be able to invoke a recovery procedure that repairs the damage and restarts the file service. In the monolithic kernel approach if the file service fails, the kernel usually fails too, and the entire operating system must reboot.

Few widely used operating systems implement the microkernel approach in its purest form. In fact, most widely used operating systems today have a mostly monolithic kernel. Many critical services run inside the kernel, and only a few run outside the kernel. For example, in the GNU/Linux operating system the file and the network service run in kernel mode, but the X Window System runs in user mode.

Monolithic operating systems dominate the field for several reasons. First, if a service (e.g., a file service) is critical to the functioning of the operating system, it doesn't matter much if it fails in user mode or in kernel mode; in either case, the system is unusable.

Second, some services are shared among many modules, and it can be easier to implement these services as part of the kernel program, which is already shared among all modules. For example, a cache of recently accessed file data is more effective when shared among all modules. Furthermore, this cache may need to coordinate its memory use with the memory manager, which is typically part of the kernel.

Third, the performance of some services is critical, and the overhead of SEND and RECEIVE supervisor calls may be too large to split subsystems into smaller modules and separate each module.

Fourth, monolithic systems can enjoy the ease of debugging microkernel systems if the monolithic kernel comes with good kernel debugging tools.

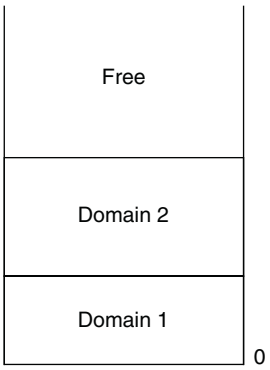
Fifth, it may be difficult to reorganize existing kernel programs. In particular, there is little incentive to change a kernel program that already works. If the system works and most of the errors have been eradicated, the debugging advantage of microkernels begins to evaporate, and the cost of SEND and RECEIVE supervisor calls begins to dominate.

In general, if one has the choice between a working system and a better designed, but new system, one doesn't want to switch over to the new system unless it is much better. One reason is the overhead of switching: learning the new design, reengineering the old system to use the new design, rediscovering undocumented assumptions, and discovering unrealized assumptions (large systems often work for reasons that weren't fully understood). Another reason is the uncertainty of the gain of switching. Until there is evidence from the field, the claims about the better design are speculative. In the case of operating systems, there is little experimental evidence that microkernel-based systems are more robust than existing monolithic kernels. A final reason is that there is an opportunity cost: one can spend time reengineering existing software, or one can spend time developing the existing software to address new needs. For these reasons, few systems have switched to a pure microkernel design. Instead many existing systems have stayed with monolithic kernels, perhaps running services that are not as performance critical as user-mode programs. Microkernel designs exist in more specialized areas, and research on microkernels continues to be active.

5.4 VIRTUALIZING MEMORY

To address one problem at a time, the previous section assumed that memory and its address space is very large, large enough to hold all domains. In practice, memory and address space are limited. Thus, when a programmer invokes ALLOCATE_DOMAIN, we would like the programmer to specify a reasonable size. To allow a program to grow its domain if the specified size turns out to be too small, we could offer the programmer an additional primitive GROW_DOMAIN.

Growing domains, however, creates memory management problems. For example, assume that program A allocates domain 1 and program 2 allocates domain 2, right after domain 1. Even if there is free memory after domain 2, program A cannot grow domain 1 because it would cross into domain 2. In this case, the only option left for program A is to allocate a new domain of the desired size, copy the contents of domain 1 into the new domain, change the addresses in the program that refer to addresses in domain 1 to instead refer to corresponding addresses in the new domain 2, and deallocate domain 1.



This memory management complicates writing programs and can make programs slow because of the memory copies. To reduce the programming burden of managing memory, most modern computer systems virtualize memory, a step that provides two features:

1. *Virtual addresses*. If programs address memory using virtual addresses and the memory manager translates the virtual addresses to physical addresses on the fly, then the memory manager can grow and move domains in memory behind the program's back.

2. *Virtual address spaces*. A single address space may not be large enough to hold all addresses of all applications at the same time. For example, a single large database program by itself may need all the address space available in the hardware. If we can create virtual address spaces, then we can give each program its own address space. This extension also allows a thread to have its program loaded at an address of its choosing (e.g., address 0).

A memory manager that virtualizes memory is called a *virtual memory manager*. The design we work out in this section replaces the domain manager but incorporates the main features of domains: controlled sharing and permissions. We describe the virtual memory design in two steps. For the first step, Sections 5.4.1 and 5.4.2 introduce virtual addresses and describe an efficient way to translate them. For the second step, Section 5.4.3 introduces virtual address spaces. Section 5.4.4 discusses the trade-offs of software and hardware aspects of implementing a virtual memory manager. Finally, the section concludes with an advanced virtual memory design.

5.4.1 Virtualizing Addresses

The virtual memory manager will deal with two types of addresses, so it is convenient to give them names. The threads issue *virtual addresses* when reading and writing to memory (see Figure 5.16). The memory manager translates each virtual address issued by the processor into a *physical address*, a bus address of a location in memory or a register on a controller of a device.

Translating addresses as they are being used provides design flexibility. One can design computers whose physical addresses have a different width than its virtual addresses. The memory manager can translate several virtual addresses to the same physical address, but perhaps with different permissions. The memory manager can allocate virtual addresses to a thread but postpone allocating physical memory until the thread makes a reference to one of the virtual addresses.

Virtualizing addresses exploits the design principle ***decouple modules with indirection***. The virtual memory manager provides a layer of indirection between the processor and the memory system by translating the virtual addresses of program instructions into physical addresses, instead of having the program directly issue physical memory addresses. Because it controls the translation from the addresses issued by the program to the addresses understood by the memory system, the virtual memory manager can translate any particular virtual address to different physical memory addresses at different times. Thanks to the translation, the virtual memory manager can rearrange the data in the memory system without having to modify any application program.

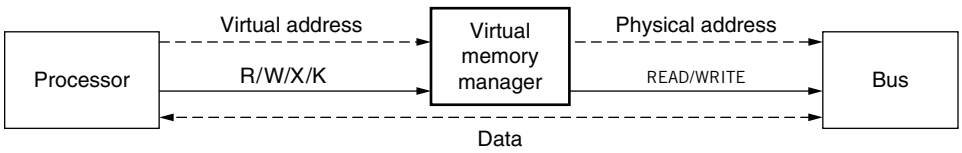


FIGURE 5.16

A virtual memory manager translating virtual addresses to physical addresses.

From a naming point of view, the virtual memory manager creates a name space of virtual addresses on top of a name space of physical addresses. The virtual memory manager's naming scheme translates virtual addresses into physical addresses.

Virtual memory has many uses. In this chapter, we focus on managing physical memory transparently. Later, in Section 6.2 of Chapter 6, we describe how virtual memory can also be used to transparently simulate a larger memory than the computer actually possesses.

To see how address translation can help memory management, consider a virtual memory manager with a virtual address space that is very large (e.g., 2^{64} bytes) but with a physical address space that is smaller. Let's assume that a thread has allocated two domains of size 100 bytes (see Figure 5.17a). The memory manager allocated the domains in physical memory contiguously, but in the virtual address space the domains are far away from each other. (`ALLOCATE_DOMAIN` returns a virtual address.) When a thread makes a reference to a virtual address, the virtual memory manager translates the address to the appropriate physical address.

Now consider the thread requesting to grow domain 1 from size 8 kilobytes to, say, 16 kilobytes. Without virtual addresses, the memory manager would deny this request because the domain cannot grow in physical memory without running into domain 2. With virtual addresses (see Figure 5.17b), however, the memory manager can grow the domain in the virtual address space, allocate the requested amount of physical memory, copy the content of domain 1 into the newly allocated physical memory, and update its mapping for domain 1. With virtual addresses, the application doesn't have to be aware that the memory manager moved the contents of its domain in order to grow it.

Even ignoring the cost of copying the content of a domain, introducing virtual addresses comes at a cost in complexity and performance. The memory manager must manage virtual addresses in addition to a physical address space. It must allocate

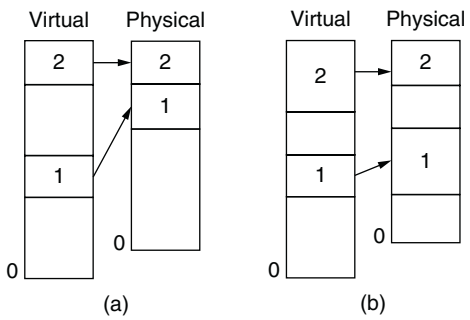


FIGURE 5.17

(a) A thread has allocated a domain 1 and 2; they are far apart in virtual memory but next to each other in physical memory. (b) In response to the thread's request to grow domain 1, the virtual memory manager transparently moved domain 1 in physical memory and adjusted the translation from virtual to physical addresses.

and deallocate them (if the virtual address space isn't large), it must set up translations between virtual and physical addresses, and so on. The translation happens on-the-fly, which may slow down memory references. The rest of this section investigates these issues and presents a plan that doesn't even require copying the complete content of a domain when growing the domain.

5.4.2 Translating Addresses Using a Page Map

A naïve way of translating virtual addresses into physical addresses is to maintain a map that for each virtual address records its corresponding physical address. Of course, the amount of memory required to maintain this map would be large. If each physical address is a word (8 bytes) and the address space has 2^{64} virtual addresses, then we might need 2^{72} bytes of physical memory just to store the mapping.

A more efficient way of translation is using a *page map*. The page map is an array of page map entries. Each entry translates a fixed-sized range of contiguous bytes of virtual addresses, called a *page*, to a range of physical addresses, called a *block*, which holds the page. For now, the memory manager maintains a single page map, so that all threads share the single virtual address space, as before.

With this organization, we can think of the memory that threads see as a set of contiguous pages. A virtual address then is a name overloaded with structure consisting of two parts: a page number and a byte offset within that page (see Figure 5.18). The page number uniquely identifies an entry in the page map, and thus a page, and the byte offset identifies a byte within that page. (If the processor provides word addressing instead of byte addressing, *offset* would specify the word within a page.) The size of a page, in bytes, is equal to the maximum number of different values that can be stored in the byte offset field of the virtual address. If the offset field is 12 bits wide, then a page contains 4,096 (2^{12}) bytes.

With this arrangement, the virtual memory manager records in the page map, for each page, the block of physical memory that contains that page. We can think of a *block* as the container of a page. Physical memory is then a contiguous set of blocks, holding pages, but the pages don't have to be contiguous in physical memory; that is, block 0 may hold page 100, block 1 may hold page 2, and so forth. The mapping between pages and the blocks that hold them can be arbitrary.

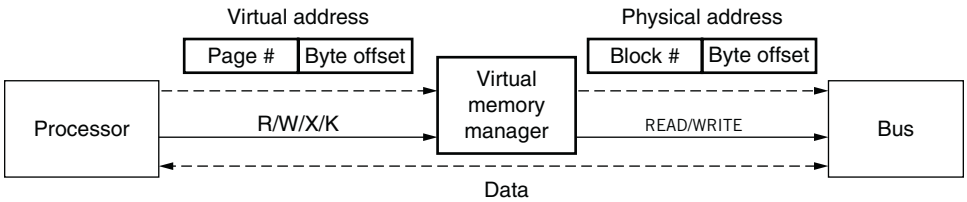


FIGURE 5.18

A virtual memory manager that translates virtual addresses by translating page numbers to block numbers.

The page map simplifies memory management because the memory manager can allocate a block anywhere in physical memory and insert the appropriate mapping into the page map, without having to copy domains in physical memory to coalesce free space.

A physical address can also be viewed as having two parts: a block number that uniquely identifies a block of memory and an offset that identifies a byte within that block. Translating a virtual address to a physical address is now a two-step process:

1. The virtual memory manager translates the page number of the virtual address to a block number that holds that page by means of some mapping from page numbers to block numbers.
2. The virtual memory manager computes the physical address by concatenating the block number with the byte offset from the original virtual address.

Several different representations are possible for the page map, each with its own set of trade-offs for translating addresses. The simplest implementation of a page map is an array implementation, often called a *page table*. It is suitable when most pages have an associated block. Figure 5.19 demonstrates the use of a page map implemented as a linear page table. The virtual memory manager resolves virtual addresses into physical addresses by taking the page number from the virtual address and using it as an index into the page table to find the corresponding block number. Then, the manager computes the physical address by concatenating the byte offset with the

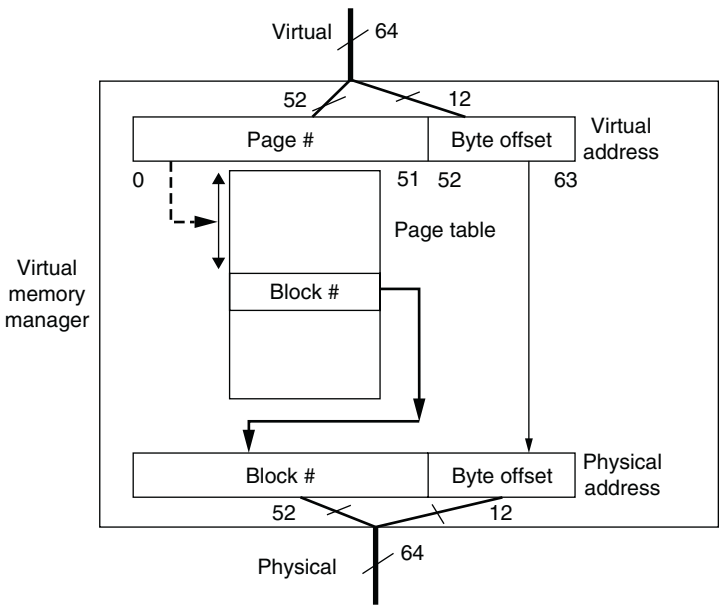


FIGURE 5.19

An implementation of a virtual memory manager using a page table.

block number found in the page-table entry. Finally, it sends this physical address to the physical memory.

If the page size is 2^{12} bytes and virtual addresses are 64 bits wide, then a linear page table is large ($2^{52} \times 52$ bits). Therefore, in practice, designers use a more efficient representation of a page map, such as a two-level one or an inverted one (i.e., indexed by physical address instead of virtual), but these designs are beyond the scope of this text.

To be able to perform the translation, the virtual memory manager must have a way of finding and storing the page table. In the usual implementation, the page table is stored in the same physical memory that holds the pages, and the *physical* address of the base of the page map is stored in a reserved processor register, typically named the *page-map address register*. To ensure that user-level threads cannot change translation directly and bypass enforced modularity, processor designs allow threads to write the page-map address register only in kernel mode and allow only the kernel to modify the page table directly.

Figure 5.20 shows an example of how a kernel could use the page map. The kernel has allocated a page map in physical memory at address 0. The page map provides modules with a contiguous universal address space, without forcing the kernel to allocate blocks of memory for a domain contiguously. In this example, block 100 contains page 12 and block 500 contains page 13. When a thread asks for a new domain or to grow an existing domain, the kernel can allocate any unused block and insert it in the page map. The level of indirection provided by the page map allows the kernel to do this transparently—the running threads are unaware.

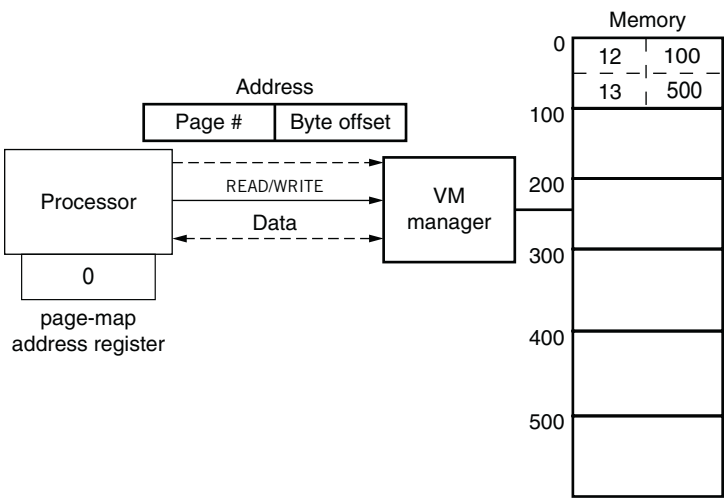


FIGURE 5.20

A virtual memory manager using a page table. The page table is located at physical address 0. It maps pages (e.g., 12) to blocks (e.g., 100).

5.4.3 Virtual Address Spaces

The design so far has assumed that all threads share a single virtual address space that is large enough that it can hold all active modules and their data. Many processors have a virtual address space that is too small to do that. For example, many processors use virtual addresses that are 32 bits wide and thus have only 2^{32} addresses, which represent 4 gigabytes of address space. This might be barely large enough to hold the most frequently used part of a large database, leaving little room for other modules. We can eliminate this assumption by virtualizing the physical address space.

5.4.3.1 Primitives for Virtual Address Spaces

A *virtual address space* provides each application with the illusion that it has a complete address space to itself. The virtual memory manager can implement a virtual address space by giving each virtual address space its own page map. A memory manager supporting multiple virtual address spaces may have the following interface:

- *id* ← CREATE_ADDRESS_SPACE (): create a new address space. This address space is empty, meaning that none of its virtual pages are mapped to real memory. CREATE_ADDRESS_SPACE returns an identifier for that address space.
- *block* ← ALLOCATE_BLOCK (): ask the memory manager for a block of memory. The manager attempts to allocate a block that is not in use. If there are no free blocks, the request fails. ALLOCATE_BLOCK returns the physical address of the block.
- MAP (*id, block, page_number, permission*): put a block into *id*'s address space. MAP maps the physical address *block* to virtual page *page_number* with permissions *permission*. The memory manager allocates an entry in the page map for address space *id*, mapping the virtual page *page_number* to block *block*, and setting the page's permissions to *permission*.
- UNMAP (*id*, *page_number*): remove the entry for *page_number* from the page map so that threads have no access to that page and its associated block. An instruction that refers to a page that has been deleted is an illegal instruction.
- FREE_BLOCK (*block*): add the block *block* to the list of free memory blocks.
- DELETE_ADDRESS_SPACE (*id*): destroy an address space. The memory manager frees the page map and its blocks of address space *id*.

Using this interface, a thread may allocate its own address space or share its address space with other threads. When a programmer calls ALLOCATE_THREAD, the programmer specifies the address space in which the thread is to run. In many operating systems, the word "*process*" is used for the combination of a single virtual address space shared by one or more threads, but not consistently (see Sidebar 5.4).

The virtual address space is a thread's domain, and the page map defines how it resides in physical memory. Thus the kernel doesn't have to maintain a separate domain table with domain registers. If a physical block doesn't appear in an address space's page map, then the thread cannot make a reference to that physical block.

If a physical block appears in an address space's page map, then the thread can make a reference to that physical block. If a physical block appears in two page maps, then threads in both address spaces can make references to that physical block, which allows sharing of memory.

The memory manager can support domain permissions by placing the permission bits in the page-map entries. For example, one address space may have a block mapped with READ and WRITE permissions, while another address space has only READ permissions for that block. This design allows us to remove the domain registers, while keeping the concept of domains.

Figure 5.21 illustrates the use of several address spaces. It depicts two threads, each with its own address space but sharing block 800. Threads A and B have block 800 mapped at page 12. (In principle, the threads could map block 800 at different virtual

Sidebar 5.4 Process, Thread, and Address Space The operating systems community uses the word "process" often, but over the years it has come up with enough variants on the concept that when you read or hear the word you need to guess its meaning from its context. In the UNIX system (see Section 2.5), a process may mean one thread in a private address space (as in the early version of that system), or a group of threads in a private address space (as in later versions), or a thread (or group of threads) in an address space that is partly or completely shared (as in later versions of UNIX that also allow processes to share memory). That range of meanings is so broad as to render the term less than useful, which is why this text uses process only in the context of the early version of the UNIX system and otherwise uses only the terms *thread* and *address space*, which are the two core concepts.

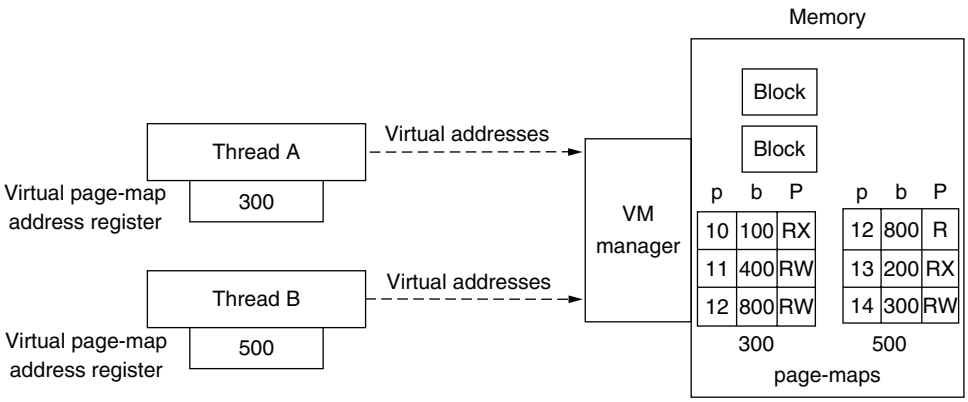


FIGURE 5.21

Each thread has its own page-map address register. Thread A runs with the page map stored at address 300, while B runs with the page map stored at address 500. The page map contains the translation from page (p) to block (b), and the permissions required (P).

addresses, but that complicates naming of the shared data in the block.) Thread A maps block 800 with READ permission, while thread B maps block 800 with READ and WRITE permissions. In addition to a shared block, each thread has two private blocks. Each thread has a block mapped with READ and EXECUTE permissions for, for example, its program text and a block mapped with READ and WRITE permissions for, for example, its stack.

To support virtual address spaces, the page-map address register of a processor holds the physical address of the page map of the running thread on the processor and translation works then as follows:

```
1  procedure TRANSLATE (integer virtual, perm_needed) returns physical_address
2      page ← virtual[0:41]                          // Extract page number
3      offset ← virtual[42:63]                       // Extract offset
4      page_table ← PMAR                             // Use the current page table
5      perm_page ← page_table[page].permissions      // Lookup permissions for page
6      if PERMITTED (perm_needed, perm_page) then
7          block ← page_table[page].address          // Index into page map
8          physical ← block + offset                 // Concatenate block and offset
9          return physical                           // Return physical address
10     else return error
```

Although usually implemented in hardware, in pseudocode form, we can view the linear page table as an array that is indexed by a page number and that stores the corresponding block number. Line *2* extracts the page number, *page*, by extracting the leftmost 42 bits. (As explained in Sidebar 4.3, this book uses the big-endian convention for numbering bits and begins numbering with zero.) Then, it extracts the *offset*, the 12 rightmost bits of the virtual address (line *3*). Line *4* reads the address from the active page map out of PMAR. Line *5* looks up the permissions for the page. If the permissions necessary for using *virtual* are a subset of the permissions for the page (line *6*), then TRANSLATE looks up the corresponding block number by using *page* as an index into *page_table* and computes the physical address by concatenating *block* with *offset* (lines *7* and *8*). Now the virtual memory manager issues the bus request for the translated physical address or interrupts the processor with an illegal memory reference exception.

5.4.3.2 The Kernel and Address Spaces

There are two options for setting up page maps for the kernel program. The first option is to have each address space include a mapping of the kernel into its address space. For example, the top half of the address space might contain the kernel, in which case the bottom half contains the user program. With this setup, switching from the user program to the kernel (and vice versa) doesn't require changing the processor's page-map address register; only the user-mode bit must be changed. To protect the kernel, the kernel sets the permissions for kernel pages to KERNEL-ONLY; in user mode, performing a STORE to kernel pages is an illegal instruction. An additional advantage of this design is that in kernel mode, it is easy for the kernel to read data structures of the user program because the user program and kernel share the same address space. A disadvantage of this design is that it reduces the available address

space for user programs, which could be a problem in a legacy architecture that has small address spaces.

The second option is for the memory manager to give the kernel its own separate address space, which is inaccessible to user-level threads. To implement this option, we must extend the SVC instruction to switch the page-map address register to the kernel's page map when entering kernel mode. Similarly, when returning from kernel mode to user mode, the kernel must change the page-map address register to the page map of the thread that entered the gate.

The second option separates the kernel program and user programs completely, but the memory manager, which is part of the kernel, must be able to create new address spaces for user programs, and so on. The simple solution is to include the page tables of all user address spaces in the kernel address space. By modifying the page table for a user address space, the memory manager can modify that address space. Since a page table is smaller than the address space it defines, the second option wastes less address space than the first option.

If the kernel program and user programs have their own address spaces, the kernel cannot refer to data structures in user programs using kernel virtual addresses, since those virtual addresses refer to locations in the kernel address space. User programs must pass arguments to supervisor calls by value or the kernel must use a more involved method for copying data from a user address space to a kernel address space (and vice versa). For example, the kernel can compute the physical address for a user virtual address using the page table for that user address space, map the computed physical address into the kernel address space at an unused address, and then use that address.

5.4.3.3 Discussion

In the design with many virtual address spaces, virtual addresses are relative to an address space. This property has the advantage that programs don't have to be compiled to be position independent (see Sidebar 5.5). Every program can be stored at virtual address 0 and can use absolute addresses for making references to memory in its address space. In practice, this advantage is unimportant because it is not difficult for compiler designers to generate position-independent instructions.

A disadvantage of the design with many address spaces is that sharing can be confusing and less flexible. It can be confusing because a block to be shared can be mapped by threads into two different address spaces at different virtual addresses.

Sidebar 5.5 Position-Independent Programs Position-independent programs can be loaded at any memory address. To provide this feature, a compiler translating programs into processor instructions must generate relative, but not absolute, addresses. For example, when compiling a **for** loop, the compiler should use a jump instruction with an offset relative to the current PC to return to the top of a **for** loop rather than a jump instruction with an absolute address.

It can be less flexible because either threads share a complete address space or a designer must accept a restriction on sharing. Threads in different address spaces can share objects only at the granularity of a block: if two threads in different address spaces share an object, that object must be mapped at a page boundary, and holding the object requires allocating an integral number of pages and blocks. If the shared object is smaller than a page, then part of the address space and the block will be wasted. Section 5.4.5 describes an advanced design that doesn't have this restriction, but it is rarely used, since the waste isn't a big problem in practice.

5.4.4 Hardware versus Software and the Translation Look-Aside Buffer

An ongoing debate between hardware and software designers concerns what parts of the virtual memory manager should be implemented in hardware as part of the processor and what parts in software as part of the operating system, as well as what the interface between the hardware module and the software module should be.

There is no "right" answer because the designers must make a trade-off between performance and flexibility. Because address translation is in the critical path of processor instructions that use addresses, the memory manager is often implemented as a digital circuit that is part of the main processor so that it can run at the speed of the processor. A complete hardware implementation, however, reduces the opportunities for the operating system to exploit the translation between virtual and physical addresses. This trade-off must be made with care when implementing the memory manager and its page table.

The page table is usually stored in the same memory as the data, reachable over the bus. This design requires that the processor make an additional bus reference to memory each time it interprets an address: the processor must first translate the virtual address into a physical address, which requires reading an entry in the page map.

To avoid these additional bus references for translating virtual to physical addresses, the processor typically maintains a cache of entries of the page map in a smaller fast memory within the processor itself. The hope is that when the processor executes the next instruction, it will discover a previously cached entry that can translate the address, without making a bus reference to the larger memory. Only when the cache memory doesn't contain the appropriate entry must the processor retrieve an entry from memory. In practice, this design works well because most programs exhibit locality of reference. Thus, caching translation entries pays off, as we will see when we study caches in Chapter 6. Caching page table entries in the processor introduces new complexities: if a processor changes a page table entry, the cached versions must be updated too, or invalidated.

A final design issue is how to implement the cache memory of translations efficiently. A common approach is to use an associative memory instead of an indexed

memory. By making the cache memory associative, any entry can store any translation. Furthermore, because the cache is much smaller than physical memory, an associative memory is feasible. In this design, the cache memory of translations is usually referred to as the *translation look-aside buffer (TLB)*.

In the hardware design of Figure 5.19, the format of the page table is determined by the hardware. RISC processors typically don't fix the format of the page table in hardware but leave the choice of data structure to software. In these RISC designs, only the TLB is implemented in hardware. When a translation is not in the TLB, the processor generates a *TLB miss exception*. The handler for this interrupt looks up the mapping in a data structure implemented in software, inserts the translation in the TLB, and returns from the interrupt. With this design, the memory manager has complete freedom in choosing the data structure for the page map. If a module uses only a few pages, a designer may be able to save memory by storing the page map as a linked list or tree of pages. If, as is common, the union of virtual addresses is much larger than the physical memory, a designer may be able to save memory by inverting the page map and storing one entry per physical memory block; the contents of the entry identify the number of the page currently in the block.

In almost all designs of virtual addresses, the operating system manages the content of the page map in software. The hardware design may dictate the format of the table, but the kernel determines the values stored in the table entries and thus the mapping from virtual addresses to physical addresses. By allowing software to control the mapping, designers open up many uses of virtual addresses. One use is to manage physical memory efficiently, avoiding problems due to fragmentation. Another use is to extend physical memory by allowing pages to be stored on other devices, such as magnetic disks, as explained in Section 6.2.

5.4.5 Segments (Advanced Topic)

An address space per program (as in Figure 5.21) limits the way objects can be shared between threads. An alternative way is to use *segments*, which provide each object with a virtual address space starting at 0 and ending at the size of the object. In the segment approach, a large database program may have a segment for each table in a database (assuming the table isn't larger than a segment's address space). This allows threads to share memory at the granularity of objects instead of blocks, and in a flexible manner. A thread can share one object (segment) with one thread and another object (segment) with another thread.

To support segments, the processor must be modified because the addresses that programs use are really two numbers. The first identifies the segment number, and the second identifies the address within that segment. Unlike the model that has one virtual address space per program, where the programmer is unaware that the virtual address is implemented as a page number and an offset, in the segment model, the compiler and programmer must be aware that an address contains two parts. The programmer must specify which segment to use for an instruction, the compiler must

put the generated code in the right segment, and so on. Problem set *11* explores segments with a simple operating system and a processor with minimal support for segments.

In the address space per program, a thread can do arithmetic on addresses because the program's address space is linear. In the segment model, a thread cannot do arithmetic on addresses in different segments because adding to a segment number yields a meaningless result; there is no notion of contiguity for segment numbers.

If two threads share an object, they typically use the same segment number for the object; otherwise naming shared objects becomes cumbersome too.

Segments can be implemented by reintroducing slightly modified domain registers. Each segment has its own domain register, but we add a *page_table* field to the domain register. This *page_table* field contains the physical address of the page table that should be used to translate virtual addresses of the segment. When domain registers are used in this way, the literature calls them *segment descriptors*. Using this implementation, the memory manager translates an address as follows: the memory manager uses the segment number to look up the segment descriptor and uses the *page_table* in the segment descriptor to translate the virtual address within the segment to a physical address.

Giving each object of an application its own segment potentially requires a large number of segment descriptors per processor. We can solve this problem by putting the segment descriptors in memory in a *segment descriptor table* and giving each processor a single register that points to the segment descriptor table.

An advantage of the segment model is that the designer doesn't have to predict the maximum size of objects that grow dynamically during computation. For example, as the stack of a running computation grows, the virtual memory manager can allocate more pages on demand and increase the length of the stack segment. In the address space per program model, the thread's stack may grow into another data structure in the virtual address space. Then either the virtual memory manager must raise an error, or the complete stack must be moved to a place in the address space that has a large enough range of unused contiguous addresses.

The programming model that goes with a segment per object can be a good match for new programs written in an object-oriented style: the methods of an object class can be in a segment with READ and EXECUTE permissions, the data objects of an instance of that class in a segment with READ and WRITE permissions, and so on. Porting an old program to the segment model can be easy if one stores the complete program, code, and data in a single segment, but this method loses much of the advantage of the segment model because the entire segment must have READ, WRITE, and EXECUTE permission. Restructuring an old program to take advantage of multiple segments can be challenging because addresses are not linear; the programmer must modify the old program to specify which segment to use. For example, upgrading a kernel program to take advantage of segments in its internal construction is disruptive. A number of processors and kernels tried but failed (see Section 5.7).

Although virtual memory systems supporting segments have advantages and have been influential (see, for example, the Multics virtual memory design [Suggestions

for Further Reading 5.4.1]), most virtual memory systems today follow the address space per program approach instead of the segment approach. A few processors, such as the Intel x86 (see Section 5.7), have support for segments, but today's virtual memory systems don't exploit them. Virtual memory managers for the address space per program model tend to be less complex because sharing is not usually a primary requirement. Designers view an address space per program primarily as a method for achieving enforced modularity, rather than an approach to sharing. Although one can share pages between programs, that isn't the primary goal, but it is possible to do it in a limited way. Furthermore, porting an old application to the one address space per program model requires little effort at the outset: just allocate a complete address space for the application. If any sharing is necessary, it can be done later. In practice, sharing patterns tend to be simple, so no sophisticated support is necessary. Finally, today, address spaces are usually large enough that a program doesn't need an address space per object.

5.5 VIRTUALIZING PROCESSORS USING THREADS

In order to focus on one new idea at a time, the previous sections assumed that a separate processor was available to run each thread. Because there are usually not enough processors to go around, this section extends the thread manager to remove the assumption that each thread has its own processor. This extended thread manager shares a limited number of processors among a larger number of threads.

Sharing of processors introduces a new concern: if a thread hogs a processor, either accidentally or intentionally, it can slow down or even halt the progress of other threads, thereby compromising modularity. Because we have proposed that a primary requirement be to enforce modularity, one of the design challenges of the thread manager is to eliminate this concern.

This section starts with the design of a simple thread manager that does not avoid the hogging of a processor, and then moves to a design that does. It makes the design concrete by providing a pseudocode implementation of a thread manager. This implementation captures the essence of a thread manager. In practice, thread managers differ in many details and sometimes are much more complex than our example implementation.

5.5.1 Sharing a Processor Among Multiple Threads

Recall from Section 5.1.1 that a thread is an abstraction that encapsulates the state of a running module. A thread encapsulates enough state of the interpreter (e.g., a processor) that executes the thread that a thread manager can stop the thread and resume the thread later. A thread manager animates a thread by giving it a processor. This section explains how this ability to stop a thread and later resume it can be used to multiplex many threads over a limited number of physical processors.

To make the thread abstraction concrete, the thread manager might support this simple version of a THREAD_ALLOCATE procedure:

- *thread_id* ← ALLOCATE_THREAD (*starting_procedure, address_space_id*): allocate a new thread in address space *address_space_id*. The new thread is to begin with a call to the procedure specified in the argument *starting_procedure*. ALLOCATE_THREAD returns an identifier that names the just-created thread. If the thread manager cannot allocate a new thread (e.g., it doesn't have enough free memory to allocate a new stack), ALLOCATE_THREAD returns an error.

The thread manager implements ALLOCATE_THREAD as follows: it allocates a range of memory in *address_space_id* to be used as the stack for procedure calls, selects a processor, and sets the processor's PC to the address *starting_procedure* in *address_space_id* and the processor's SP to the bottom of the allocated stack.

Using ALLOCATE_THREAD, an application can create more threads than there are processors. Consider the applications running on the computer in Figure 5.4. These applications can have more threads than there are processors; for example, the file service might launch a new thread for each new client request. Starting a new module will also create additional threads. So the problem is to share a limited number of processors among potentially many threads.

We can solve this problem by observing that most threads spend much of their time waiting for events to happen. Most modules that run on the computer in Figure 5.4 call READ for input from the keyboard, the file system, or the network, and will wait by spinning until there is input. Instead of spinning, the thread could, while it is waiting, let another thread use its processor. If the consumer thread finds that it cannot proceed (because the buffer is empty), then it could release its processor, giving the keyboard manager (or any other thread) a chance to run.

This observation is the basic idea for virtualizing the processor: when a thread is waiting for an event, its processor can switch from that thread to another one by saving the state of the waiting thread and loading the state of a different thread. Since in most system designs many threads spend much of their life waiting for a condition to become true, this idea is general. For example, most of the other modules (the window manager, the mail reader, etc.) are consumers. They spend much of their existence waiting for input to arrive.

Over the years people have developed various labels for processor virtualization schemes, such as "time-sharing", "processor multiplexing", "multiprogramming", or "multitasking". For example, the word "time-sharing" was introduced in the 1950s to describe virtualization of a computer system so that it could be shared among several interactive human users. All these schemes boil down to the same idea: virtualizing the processor, which this section describes in detail.

To make the discussion more concrete, consider the implementation of SEND and RECEIVE with a bounded buffer in Figure 5.6. This spin-loop solution is appropriate if there is a processor for each thread, but it is inappropriate if there are fewer processors than threads. If there is just one processor and if the receiver started before the sender, then we have a major problem. The receiver thread executes its

spinning loop, and the sender never gets a chance to run and add an item to the buffer.

A solution with thread switching is shown in Figure 5.22. Comparing this code with the code in Figure 5.6, we find that the only change is the addition of two calls to a procedure named YIELD (lines *10* and *20*). YIELD is an entry to the thread manager. When a thread invokes YIELD, the thread manager gives the calling thread's processor to some other thread. At some time in the future, the thread manager returns a processor to this thread by returning from the call to YIELD. In the case of the receiver, when the processor returns at line *21*, the receiving thread reacquires the lock again and tests *out* = *in*. If *out* is now less than *in*, there is at least one new item in the buffer, so the thread extracts an item from the buffer. If not, the thread releases the lock and calls YIELD again to allow another thread to run. A thread therefore alternates between two states, which we name RUNNING (executing on a processor) and RUNNABLE (ready to run but waiting for a processor to become available).

```
1   shared structure buffer                  // A shared bounded buffer
2       message instance message[N]          // with a maximum of N messages
3       long integer in initially 0          // Counts number of messages put in the buffer
4       long integer out initially 0         // Counts number of messages taken out of the buffer
5       lock instance buffer_lock initially UNLOCKED // Lock to coordinate sender and receiver

6   procedure SEND (buffer reference p, message instance msg)
7       ACQUIRE (p.buffer_lock)
8       while p.in − p.out = N do            // Wait until there room in the buffer
9           RELEASE (p.buffer_lock)          // Release lock so that receiver can remove
10          YIELD ()                         // Let another thread use the processor
11          ACQUIRE (p.buffer_lock)          // Processor came back, maybe there is room
12                                           // Wait loop end, go back to test
13      p.message[p.in modulo N] ← msg       // Put message in the buffer
14      p.in ← p.in + 1                      // Increment in
15      RELEASE (p.buffer_lock)

16  procedure RECEIVE (buffer reference p)
17      ACQUIRE (p.buffer_lock)
18      while p.in = p.out do                // Wait until there is a message to receive
19          RELEASE (p.buffer_lock)          // Release lock so that sender can add
20          YIELD ()                         // Let another thread use the processor
21          ACQUIRE (p.buffer_lock)          // YIELD returned, maybe there is a message
22                                           // Wait loop end, go back to test
23      msg ← p.message[p.out modulo N]      // Copy item out of buffer
24      p.out ← p.out + 1                    // Increment out
25      RELEASE (p.buffer_lock)
26      return msg
```

FIGURE 5.22

An implementation of a virtual communication link for a system with more threads than processors.

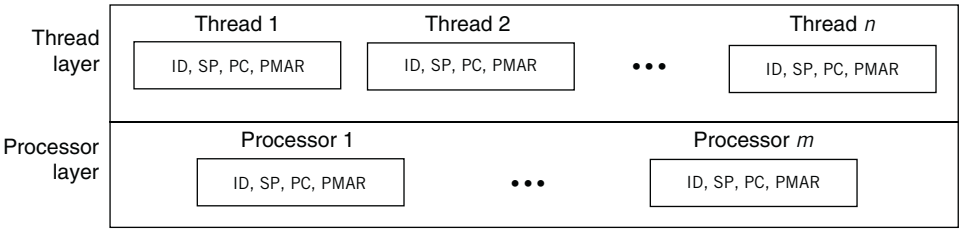


FIGURE 5.23

Multiplexing m processors among n threads ($n > m$).

The job of YIELD is to switch a processor from one thread to another. In its essence, YIELD is a simple three-step operation:

1. *Save* this thread's state so that it can resume later.
2. *Schedule* another thread to run on this processor.
3. *Dispatch* this processor to that thread.

The concrete problem that YIELD solves is multiplexing many threads over a potentially smaller number of processors (see Figure 5.23). Each processor typically has an identifier (ID), a stack pointer (SP), a program counter (PC), and a page-map address register (PMAR), pointing to the page map that defines the thread's address space. Processors may have additional state, such as floating point registers. Each thread has virtual versions of ID, SP, PC, and PMAR, and any additional state. YIELD must multiplex perhaps many threads in the thread layer over a limited number of processors in the processor layer.

YIELD implements the multiplexing as follows. When a thread running in the thread layer calls YIELD, YIELD enters the processor layer. The processor saves the state of the thread that is currently running. When that processor later exits from the processor layer, it runs a new thread, usually one that is different from the one it was running when it entered. This new thread may run in a different address space from the one used by the thread that called YIELD, or it may run in the same address space, depending on how the two threads were originally allocated.

Because the implementation of YIELD is specific to a processor and must load and save state that is often stored in processor registers (i.e., SP, PC, PMAR), YIELD is written in the instruction repertoire of the processor and can be thought of as a software extension of the processor. Programs using YIELD may be written in any programming language, but the implementation of YIELD itself is usually written in low-level processor-specific instructions. YIELD is typically a kernel procedure reached by a supervisor call.

Using this layering picture, we can also explain how interrupts and exceptions are multiplexed with threads. Interrupts invoke an interrupt handler, which always runs in the processor layer, even if the interrupt occurs while in the thread layer. On an interrupt, the interrupted processor runs the corresponding interrupt handler

(e.g., when a clock interrupt occurs, it runs the clock handler) and then continues with the thread that the processor was running before the interrupt.

Exceptions happen in the thread layer. That is, the exception handler runs in the context of the interrupted thread; it has access to the interrupted thread's state and can invoke procedures on behalf of the interrupted thread.

As discussed in Sidebar 5.6, the literature is inconsistent both about the labels and about the distinction between the concepts of interrupts and exceptions. For purposes of this text, we define interrupts as events that may have no relation to the currently running thread, whereas exceptions are events that specifically pertain to the currently running thread. While both exceptions and interrupts are discovered by the processor, interrupts are handled by the processor layer, and exceptions are usually referred to a handler in the thread layer.

This difference places restrictions on what code can run in an interrupt handler: in general, an interrupt handler shouldn't invoke procedures (e.g., YIELD) of the thread layer that assume that the thread is running on the current processor because the interrupt may have nothing to do with the currently running thread. An interrupt handler can invoke an exception handler in the thread layer if the handler determines that this interrupt pertains to the thread running on the interrupted processor. The exception handler can then adjust the thread's environment. We will see an example of this case in Section 5.5.4 when the thread manager uses a clock interrupt to force the currently running thread to invoke YIELD.

Although the essence of multiplexing is simple, the code implementing YIELD is often among the most mysterious in an operating system. To dispel the mysteries, Section 5.5.2 develops a simple implementation of a thread manager that supports YIELD. Section 5.5.3 describes how this implementation can be extended to support creation and termination of threads. Section 5.5.4 explains how an operating system can enforce modularity among threads using interrupts and Section 5.5.5 adds enforcement of separate address spaces. Section 5.5.6 explains how systems use multiplexing recursively to implement several layers of processor virtualization.

Sidebar 5.6 Interrupts, Exceptions, Faults, Traps, and Signals The systems and architecture literature uses the words "interrupt" and "exception" inconsistently, and some authors use different words, such as "fault", "trap", "signal", and "sequence break". Some designers call a particular event an interrupt, while another designer calls the same event an exception or a signal. An operating system designer may label the handler for a hardware interrupt as an exception, trap, or fault handler. Terminology questions also arise because an interrupt handler in the operating system may invoke a thread's exception handler, which raises the question of whether the original event is an interrupt or an exception. The layered model helps answer this question: at the processor layer the event is an interrupt, and at the thread layer it is an exception.

5.5.2 Implementing YIELD

To keep the implementation of YIELD as simple as possible, let's temporarily restrict its implementation to a fixed number of threads, say, seven, and assume there are fewer than seven processors. (If there were seven or more processors and only seven threads, then processor virtualization would be unnecessary.) We further start by assuming that all seven threads run in the same address space, so we don't have to worry about saving and restoring a thread's PMAR. Finally, we will assume that the threads are already running. (Section 5.5.3 will remove the last assumption, explaining how threads are created and how the thread manager starts.)

With these assumptions we can implement YIELD as shown in Figure 5.24. The implementation of yield relies on four procedures: GET_THREAD_ID, ENTER_PROCESSOR_LAYER, EXIT_PROCESSOR_LAYER, and SCHEDULER. Each procedure has only a few lines of code, but they are subtle; we investigate them in detail.

As shown in the figure, the code for the procedures maintains two shared arrays: an array with one entry per processor, known as the *processor_table*, and an array with one entry per thread, known as the *thread_table*. The *processor_table* array records information for each processor. In this simple implementation, the information is just the identity of the thread that the processor is currently running. In later versions, we will need to keep track of more information. To be able to index into this table, a processor needs to know what its identity is, which is usually stored in a special register CPUID. That is, the procedure GET_THREAD_ID returns the identity of the thread running on processor CPUID (line *7*). The procedure GET_THREAD_ID virtualizes the register CPUID to create a virtual ID register for each thread, which records a thread's identity.

Entry *i* of *thread_table* holds the stack pointer for thread *i* (whenever thread *i* is not actually running on a processor) and records whether thread *i* is RUNNING (i.e., a processor is running thread *i*) or RUNNABLE (i.e., thread *i* is waiting to receive a processor). In a system with n processors, n threads can be in the RUNNING state at the same time.

With these data structures, the processor layer works as follows. Suppose that two processors, A and B, are busy running seven threads and that thread 0, which is running on processor A, calls YIELD. YIELD acquires *thread_table_lock* at line *9* so that the processor layer can implement switching threads as a before-or-after action. (The lock is needed because there is more than one processor, so different threads might try to invoke YIELD at the same time.) YIELD then calls ENTER_PROCESSOR_LAYER to release its processor.

The statement on line *14* records that the calling thread will no longer be running on the processor but that it is runnable. That is, if there are no other threads waiting to run, the processor layer can schedule thread 0 again.

Line *15* saves thread 0's stack pointer (held in processor A's SP register) into thread 0's entry in *thread_table*. The stack pointer is the only thread state that must be saved because the processor layer always suspends a thread in ENTER_PROCESSOR_LAYER; it is unnecessary to save and restore the program counter. We are assuming that all threads run in the same address space so PMAR doesn't have to be saved and restored either. Other processors or calling conventions (or if a thread may be resumed at a different address than in ENTER_PROCESSOR_LAYER) might require that ENTER_PROCESSOR_LAYER save

```
1     shared structure processor_table[7]
2         integer thread_id                  // identity of thread running on a processor
3     shared structure thread_table[7]
4         integer topstack                   // value of this thread's stack pointer
5         integer state                      // RUNNABLE OR RUNNING
6     shared lock instance thread_table_lock

7     procedure GET_THREAD_ID() return processor_table[CPUID].thread_id

8     procedure YIELD ()
9         ACQUIRE (thread_table_lock)
10        ENTER_PROCESSOR_LAYER (GET_THREAD_ID())
11        RELEASE (thread_table_lock)
12        return

13    procedure ENTER_PROCESSOR_LAYER (this_thread)
14        thread_table[this_thread].state ← RUNNABLE     // switch state to RUNNABLE
15        thread_table[this_thread].topstack ← SP        // store yielding's thread SP
16        SCHEDULER()
17        return

18    procedure SCHEDULER()
19        j = GET_THREAD_ID()
20        do                                             // schedule a RUNNABLE thread
21            j ← (j + 1) modulo 7
22        while thread_table[j].state ≠ RUNNABLE         // skip running threads
23        thread_table[j].state ← RUNNING                // set state to RUNNING
24        processor_table[CPUID].thread_id ← j           // record that processor runs thread j
25        EXIT_PROCESSOR_LAYER (j)                       // dispatch this processor to thread j
26        return

27    // EXIT_PROCESSOR_LAYER returns from the new thread's invocation of
28    // ENTER_PROCESSOR_LAYER and returns control to the new thread's invocation of YIELD.
29    procedure EXIT_PROCESSOR_LAYER (new)
30        SP ← thread_table[new].topstack                // dispatch: load SP of new thread
31        return
```

FIGURE 5.24

An implementation of YIELD. EXIT_PROCESSOR_LAYER will return to YIELD because EXIT_PROCESSOR_LAYER uses the SP that was saved in ENTER_PROCESS_LAYER. To make it easier to follow, the procedures have explicit **return** statements.

additional thread state. In that case, the *thread_table* entries must have additional fields, and ENTER_PROCESSOR_LAYER would save the additional state in the additional fields of the *thread_table* entries.

The part of the processor layer that chooses the next thread is called the *scheduler*. In our simple implementation, statements on lines *20* through *22* constitute the core of the scheduler. Processor A cycles through the thread table, skips threads that are already running on another processor, stops searching when it finds a runnable

thread (let's say thread 6), and sets thread 6's state to RUNNING (line *23*) so that another processor doesn't again select thread 6. This implementation schedules threads in a *round-robin* fashion, but many other policies are possible; we discuss some others in Chapter 6 (Section 6.3).

This implementation of the processor layer assumes that the number of threads is more than (or at least equal to) the number of processors. Under this assumption, processor A will select and run a thread different from the one that called YIELD, unless the number of threads is the same as the number of processors, in which case processor A will cycle back to the thread that called YIELD because all the other threads are running on other processors. If there are fewer threads than processors, this implementation leaves processor A cycling forever through *thread_table* without giving up *thread_table_lock*, preventing any other thread from calling YIELD. We will fix this problem in Section 5.5.3, where we introduce a version of YIELD that supports the dynamic creation and termination of threads.

After selecting thread 6 to run, the processor records that thread 6 is running on this processor (line *24*), so that on the next call to ENTER_PROCESSOR_LAYER the processor knows which thread it is running. The processor leaves the processor layer by calling EXIT_PROCESSOR_LAYER, which dispatches processor A to thread 6. This part of the thread manager is often called the *dispatcher*.

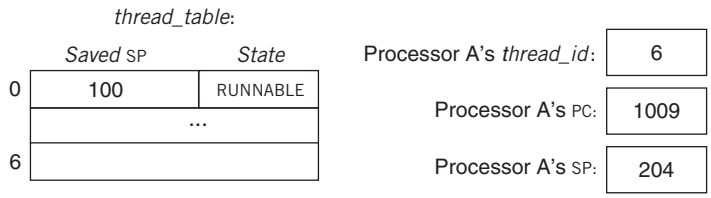
The procedure EXIT_PROCESSOR_LAYER loads the saved stack pointer of thread 6 into processor A's SP register (line *30*). (In implementations that have additional thread state that must be restored, these lines would need to be expanded.) Now processor A is running thread 6.

Because line *30* replaces SP with the value that thread 6 saved on line *15* when it last ran, the flow of control when the processor reaches the return on line *31* requires some thought. That return pops a return address off the stack. The return address is the address that thread 6 pushed on its stack when it called ENTER_PROCESSOR_LAYER at line *10*. Thus, the line *31* return actually goes to the caller of ENTER_PROCESSOR_LAYER, namely, YIELD, at line *11*. Line *12* pops the next return address off the stack, returning control to the program in thread 6 that originally called YIELD. The overall effect is that thread 0 called YIELD, but control returns to the instruction after the call to YIELD in thread 6.

This flow of control has the curious effect of abandoning two stack frames, the ones allocated on the calls to SCHEDULER and EXIT_PROCESSOR_LAYER. The original save of SP in thread 6 at line *15* actually accomplished two goals: (1) save the value of SP for future use when control returns to thread 6, and (2) mark a place that the processor layer thread can use as a stack when executing SCHEDULER and EXIT_PROCESSOR_LAYER. The reloading of SP at line *30* similarly accomplishes two goals: (1) restore the thread 6 stack and (2) abandon the processor layer stack, which is no longer needed. A more elaborate thread manager design, as we will see in Section 5.5.3, switches to a separate processor layer stack rather than borrowing space atop an existing thread layer stack.

To understand why this implementation of YIELD works, consider two threads: one running the SEND procedure of Figure 5.22 and one running the RECEIVE procedure. Furthermore, assume that the sender thread is thread 0, that the receiver thread is thread 6, and that the data and instructions of the procedures are located at address

1001 and up in memory. Finally, assume the following saved thread state for thread 0 and the following current state for processor A:



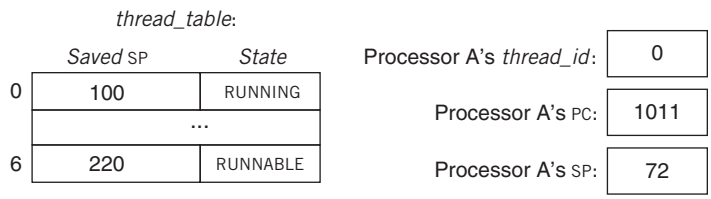
At some time in the past, thread 0 called YIELD and ENTER_PROCESSOR_LAYER stored the value of thread 0's stack pointer (100) into the thread table and went on to run some other thread. Processor A is currently running thread 6: A's entry in the *processor_table* array contains 6, A's SP register points to the top of the stack of thread 6, and A's PC register contains address 1009, which holds the first instruction of YIELD (see line *9*).

YIELD invokes the procedure ENTER_PROCESSOR_LAYER, following the procedure call convention of 4.1.1, which pushes some values on thread 6's stack—in particular, the return address (1011)—and change A's SP to 220 (204 + 16).* ENTER_PROCESSOR_LAYER knows that the current thread has index 6 by reading the processor's entry in the *processor_table* array. Line *15* saves thread 6's current top of stack (220) by storing processor A's SP into thread 6's entry into *thread_table*.

The statements at lines *19* through *22* choose which thread to run next, using a simple round-robin algorithm, and select thread 0. The scheduler invokes EXIT_PROCESSOR_LAYER to dispatch processor A to thread 0.

Line *30* loads the saved SP of thread 0 so that processor A can find the top of the stack at memory address 100. At the top of thread 0's stack will be the return address; this address will be 1011 (the line after the call to ENTER_PROCESSOR_LAYER into YIELD, line *11*), since thread 0 entered ENTER_PROCESSOR_LAYER from YIELD. Thread 0 releases *thread_table_lock* so that another thread can enter ENTER_PROCESSOR_LAYER and return from YIELD. Thread 0 returns from EXIT_PROCESSOR_LAYER following the procedure call convention, which pops off the return address from the top of the stack. The address that EXIT_PROCESSOR_LAYER uses is 1011 because EXIT_PROCESSOR_LAYER uses the SP saved by ENTER_PROCESSOR_LAYER and thus returns to YIELD at line *11*. YIELD releases the *thread_table_lock* and returns control to the program in thread 0 that originally called YIELD.

At this point, the thread switch has completed, and thread 0, rather than thread 6, is running on processor A; the state is as follows:



*The 16 bytes provide space to save R0, R1, one argument, and a return address.

At some time in the future, the thread manager will resume thread 6, at the instruction at address 1011.

From this example we can see that a thread always releases its processor by calling ENTER_PROCESSOR_LAYER and that the thread always resumes right after the call to ENTER_PROCESSOR_LAYER. This stylized flow of control in which a thread always releases its processor at the same point and resumes at that point is an example of what is sometimes called *co-routine*.

To ensure that the thread switch is atomic, the thread that invokes ENTER_PROCESSOR_LAYER acquires *thread_table_lock* and the thread that resumes using EXIT_PROCESSOR_LAYER releases *thread_table_lock* (line *11*). Because the scheduler is likely to choose a different thread to run from the one that called YIELD, the thread that releases the lock is most likely a different thread from the one that acquired the lock. In essence, the thread that releases the processor passes the lock along to the thread that next receives the processor.

Thread switching relies on a detailed understanding of the processor and the procedure call convention. In most systems, the implementation of thread switching is more complex than the implementation in Figure 5.24 because we made several assumptions that often don't hold in real systems: there is a fixed number of threads, all threads are runnable, and scheduling threads round-robin is an acceptable policy. In the next sections, we will eliminate some of these assumptions.

5.5.3 Creating and Terminating Threads

The example YIELD procedure supports only a fixed number of threads. A full-blown thread manager allows threads to be created and terminated on demand. To support a variable number of threads, we would need to modify the implementation of ALLOCATE_THREAD and extend the thread manager with the following procedures:

- EXIT_THREAD (): destroy and clean up the calling thread. When a thread is done with its job, it invokes EXIT_THREAD to release its state.
- DESTROY_THREAD (*id*): destroy the thread identified by *id*. In some cases, one thread may need to terminate another thread. For example, a user may have started a thread that turns out to have a programming error such as an endless loop, and thus the user wants to terminate it. For these cases, we might want to provide a procedure to destroy a thread.

For the most part, the implementation of these procedures is relatively straightforward, but there are a few subtle issues. For example, if threads can terminate, we have to fix the problem that the previous code required at least as many threads as processors. To get at these issues, we detail their implementation. First, we create a separate thread for each processor (which we will call a *processor-layer thread*, or *processor thread* for short), which runs the procedure SCHEDULER (see Figure 5.25). The way to think about this setup is that the SCHEDULER runs in the processor layer, and it virtualizes its processor. A processor thread per processor is necessary because a thread in the thread layer (a *thread-layer thread*) cannot deallocate its own stack since it cannot call a procedure (e.g., DEALLOCATE or YIELD) on a stack that it has released. Instead,

we set it up so that the processor-layer thread cleans up thread-layer threads. When starting the operating system kernel (e.g., after turning the computer on), the kernel creates processor-layer threads as follows:

```
procedure RUN_PROCESSORS ()
    for each processor do
        allocate stack and set up a processor thread
        shutdown ← FALSE
        SCHEDULER ()
        deallocate processor thread stack
        halt processor
```

This procedure allocates a stack and sets up a processor thread for each processor. This thread runs the scheduler procedure until some procedure sets the global variable *shutdown* to TRUE. Then, the computer restarts or halts.

We first revisit YIELD with this setup, and we then see how this generalization supports thread creation and deletion. Using a separate processor thread, we find that switching a processor from one thread-layer thread to another actually requires two thread switches: one from the thread that is releasing its processor to the processor thread and one from the processor thread to the thread that is to receive the processor (see Figure 5.26). In more detail, let's suppose, as before, that thread 0 calls YIELD on processor A and that thread 6 is runnable and has called YIELD earlier. Thread 0 switches to the processor thread by invoking ENTER_PROCESSOR_LAYER (line *12*). The implementation of ENTER_PROCESSOR_LAYER is almost identical to ENTER_PROCESSOR_LAYER of Figure 5.24: it saves the stack pointer in the calling thread's *thread_table* entry, but it loads a new stack pointer from CPUID's *processor_table* entry. When ENTER_PROCESSOR_LAYER returns, it will switch to the processor thread and resume at line *24* (right after EXIT_PROCESSOR_LAYER).

The processor thread will cycle through the thread table until it hits thread 6, which is runnable. The SCHEDULER sets thread 6's state to RUNNING (line *21*), records that thread 6 will run on this processor (line *22*), and invokes EXIT_PROCESSOR_LAYER, to switch the processor to thread 6 (line *23*). EXIT_PROCESSOR_LAYER saves the scheduler's thread state into CPUID's entry in the *processor_table* and loads thread 6's state in the processor. Because line *37* of EXIT_PROCESSOR_LAYER has loaded SP, the **return** statement at line *38* acts like a return from the procedure that saved SP. That procedure was ENTER_PROCESSOR_LAYER at line *33*, so control passes to the caller of ENTER_PROCESSOR_LAYER, namely, YIELD, at line *13*. YIELD releases *thread_table_lock* and returns control to the program of thread 6 that originally called it.

With this setup of thread switching in place, we can return to creating and deallocating threads dynamically. To keep track of which *thread_table* entries are in use, we extend the set of possible states of each entry with the additional state FREE. Now we can implement ALLOCATE_THREAD as follows:

1. Allocate space in memory for a new stack.
2. Place on the new stack an empty frame containing just a return address and initialize that return address with the address of EXIT_THREAD.

```
1     shared structure processor_table[7] // each processor maintains the following information:
2         integer topstack                // value of stack pointer
3         byte reference stack            // preallocated stack for this processor
4         integer thread_id               // identity of thread currently running on this processor
5     shared structure thread_table[7]    // each thread maintains the following information:
6         integer topstack                // value of the stack pointer
7         integer state                   // RUNNABLE, RUNNING, or FREE
8         boolean kill_or_continue        // terminate this thread? initialized to CONTINUE
9         byte reference stack            // stack for this thread

10    procedure YIELD ()
11        ACQUIRE (thread_table_lock)
12        ENTER_PROCESSOR_LAYER (GET_THREAD_ID(), CPUID)     // See caption below!
13        RELEASE (thread_table_lock)
14        return
15
16    procedure SCHEDULER ()
17        while shutdown = FALSE do
18            ACQUIRE (thread_table_lock)
19            for i from 0 until 7 do
20                if thread_table[i].state = RUNNABLE then
21                    thread_table[i].state ← RUNNING
22                    processor_table[CPUID].thread_id ← i
23                    EXIT_PROCESSOR_LAYER (CPUID, i)
24                    if thread_table[i].kill_or_continue = KILL then
25                        thread_table[i].state ← FREE
26                        DEALLOCATE(thread_table[i].stack)
27                        thread_table[i].kill_or_continue = CONTINUE
28            RELEASE (thread_table_lock)
29        return                                          // Go shut down this processor

30    procedure ENTER_PROCESSOR_LAYER (tid, processor)
31        thread_table[tid].state ← RUNNABLE
32        thread_table[tid].topstack ← SP                 // save state: store yielding's thread SP
33        SP ← processor_table[processor].topstack        // dispatch: load SP of processor thread
34        return

35    procedure EXIT_PROCESSOR_LAYER (processor, tid) // transfers control to after line 14
36        processor_table[processor].topstack ← SP    // save state: store processor thread's SP
37        SP ← thread_table[tid].topstack             // dispatch: load SP of thread
38        return
```

FIGURE 5.25

YIELD with support for dynamic thread creation and deletion. Control flow is not obvious because some of those procedures reload SP, which changes the place to which they return. To make it easier to follow, the procedures have explicit **return** statements. The procedure called on line *12* actually returns by passing control to line *24*, and the procedure called on line *23* actually returns by passing control to line *13*. Figure 5.26 shows the control flow graphically.

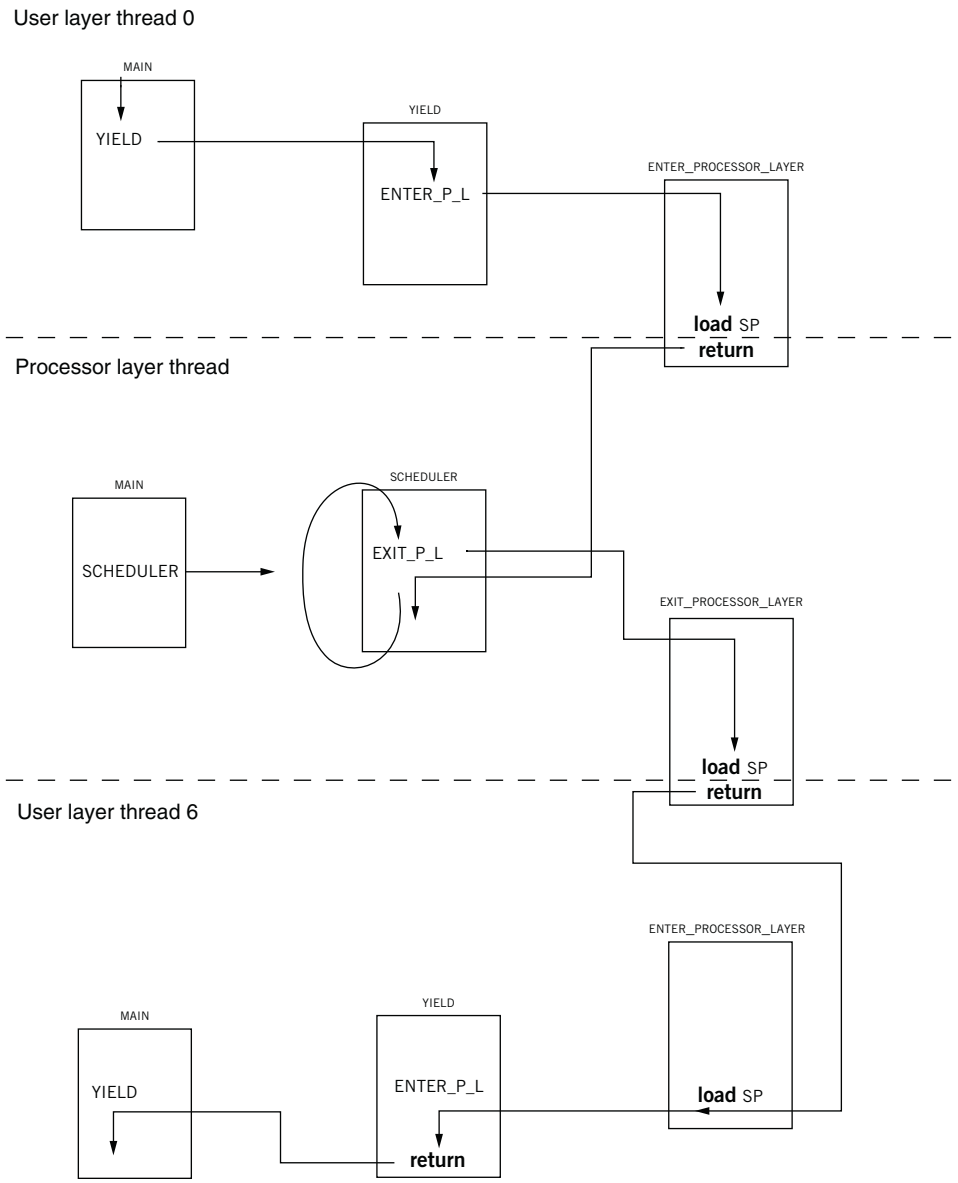


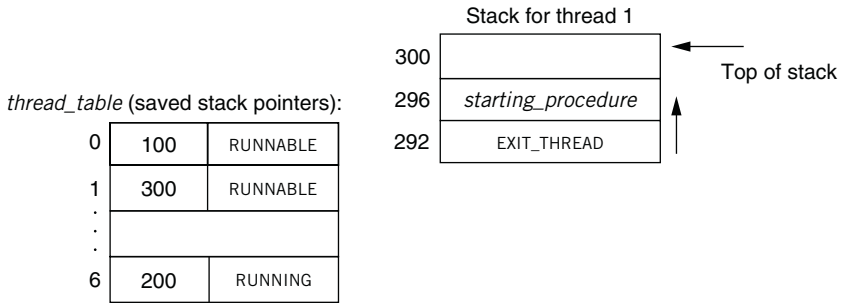
FIGURE 5.26

Control flow example when thread 0 yields to thread 6.

3. Place on the stack a second empty frame containing just a return address and initialize this return address with the address of *starting_procedure*.
4. Find an entry in the thread table that is FREE and initialize that entry for the new thread in the thread table by storing the top of the new stack.
5. Set the state of newly created thread to RUNNABLE.

If the thread manager cannot complete these steps (e.g., all entries in the thread table are in use), then THREAD_ALLOCATE returns an error.

To illustrate this implementation, consider the following state for a newly created thread 1:



Thread 1's stack is located at address 292 and its saved stack pointer is 300. With this initial setup, it appears that EXIT_THREAD called the procedure STARTING_PROCEDURE, and thread 1 is about to return to this procedure. Thus, when SCHEDULER selects this thread, its return statement will go to the procedure *starting_procedure*. In detail, when the scheduler selects the new thread (1) as the next thread to execute, it sets its stack pointer to the top of the new stack (300) in EXIT_PROCESSOR_LAYER. When the processor returns from EXIT_PROCESSOR_LAYER, it will set its program counter to the address on top of the stack (*starting_procedure*), and start execution at that location. The procedure *starting_procedure* releases *thread_table_lock* and the new thread is running.

With this initial setup, when a thread finishes the procedure *starting_procedure*, it returns using the standard procedure return convention. Since the THREAD_CREATE procedure has put the address of the EXIT_THREAD procedure on the stack, this return transfers control to the first instruction of the EXIT_THREAD procedure.

The EXIT_THREAD procedure can be implemented as follows:

```
1  procedure EXIT_THREAD()
2      ACQUIRE (thread_table_lock)
3      thread_table[tid].kill_or_continue ← KILL
4      ENTER_PROCESSOR_LAYER (GET_THREAD_ID (), CPUID)
```

EXIT_THREAD sets the *kill_or_continue* variable for thread and calls ENTER_PROCESSOR_LAYER, which switches the processor to the processor thread. The processor thread checks the variable *kill_or_continue* on line *24* to see if a thread is done, and, if so, it

marks the thread entry as reusable (line *25*) and deallocates its stack (line *26*). Since no thread is using that stack, it is safe to deallocate it.

The implementation of DESTROY_THREAD is also a bit tricky because the target thread to be destroyed might be running on one of the processors. Thus, the calling thread cannot just free the target thread's stack; the processor running the target thread must do that. We can achieve that in an indirect way. DESTROY_THREAD just sets the *kill_or_continue* variable of the target thread to KILL and returns. When a thread invokes YIELD and enters the processor layer, the processor thread will check this variable and release the thread's resources. (Section 5.5.4 will show how to ensure that each thread running on a processor will call YIELD at least occasionally.)

The implementation described for allocating and deallocating threads is just one of many ways of handling the creation and destruction of threads. If one opens up the internals of half a dozen different thread packages, one will find half a dozen quite different ways to handle launching and terminating threads. The goal of this section was not to exhibit a complete catalog, but rather, by illustrating one example in detail, to dispel any mystery and expose the main issues that every implementation must address. Problem set *10* explores an implementation of a thread package in a trivial operating system for a single processor and two threads.

5.5.4 Enforcing Modularity with Threads: Preemptive Scheduling

The thread manager described so far switches to a new thread only when a thread calls YIELD. This scheduling policy, where a thread continues to run until it gives up its processor, is called *non-preemptive scheduling*. It can be problematic because the length of time a thread holds its processor is entirely under the control of the thread itself. If, for example, a programming error sends one thread into an endless loop, no other thread will ever be able to use that processor again. Non-preemptive scheduling might be acceptable for a single module that has several threads (e.g., a Web server that has several threads to increase performance) but not for several modules.

Some systems partially address this problem by having a gentlemen's agreement called *cooperative scheduling (*which in the literature sometimes is called *cooperative multitasking*): every thread is supposed to call YIELD periodically, for instance, once per 100 milliseconds. This solution is not robust because it relies on modules behaving well and not having any errors. If a programmer forgets to put in a YIELD, or if the program accidentally gets into an endless loop that does not include a YIELD, that processor will no longer participate in the gentlemen's agreement. If, as is common with cooperative multitasking designs, there is only a single processor, the processor may appear to freeze, since the other threads won't have an opportunity to make progress.

To enforce modularity among multiple threads, the operating system thread manager must ensure thread switching by using what is called *preemptive scheduling*. The thread manager must force a thread to give up its processor after, for example, 100 milliseconds. The thread manager can implement preemptive scheduling by setting the interval timer of a clock device. When the timer expires, the clock triggers

an interrupt, switching to kernel mode in the processor layer. The clock interrupt handler can then invoke an exception handler, which runs in the thread layer and forces the currently running thread to yield. Thus, if a thread is in an endless loop, it receives 100 milliseconds to run on its turn, but it cannot stop other threads from getting at least some use of the processor, too.

Supporting preemptive scheduling requires some changes to the thread manager because in the implementation described so far an interrupt handler shouldn't invoke procedures in the thread layer at all, not even when the interrupt pertains to the currently running thread. To see why, consider a processor that invokes an interrupt handler that calls YIELD. If the interrupt happens right after the thread on that processor has acquired *thread_table_lock* in YIELD, then we will create a deadlock. The YIELD call in the handler will try to acquire *thread_table_lock* too, but it already has been acquired by the interrupted thread. That thread cannot continue and release the lock because it has been interrupted by the handler.

The problem is that we have concurrent activity within the processor layer (see Figure 5.23): the thread manager (i.e., YIELD) and the interrupt handler. The concurrent execution within the thread layer is coordinated with locks, but the processor needs its own mechanism. The processor may stop processing instructions of a thread at any time and switch to processing interrupt instructions. We are lacking a mechanism to turn the processor instruction stream and the interrupt instruction stream into separate before-or-after actions.

One solution to prevent the interrupt instruction stream from interfering with the processor instruction stream is to enable/disable interrupts. Disabling interrupts for a region greater than the region in which the *thread_table_lock* is set ensures that both streams are separate before-or-after actions. When a thread is about to acquire the *thread_table_lock*, it also disables interrupts on its processor. Now the processor will not switch to an interrupt handler when an interrupt arrives; interrupts are delayed until they are enabled again. After the thread has released the *thread_table_lock*, it is safe to reenable interrupts. Any pending interrupts will then execute immediately, but it is now safe since no thread on this processor can be inside the thread manager. This solution avoids the deadlock problem. For a more detailed description of the challenges and the solution in the Plan 9 operating system, see Suggestions for Further Reading 5.3.5.

Problem set *9* explores an implementation of a thread package with preemptive scheduling for a trivial operating system tailored to a single processor, which allows for other solutions to coordinating interrupts.

Preemptive scheduling is the mechanism that enforces modularity among threads because it isolates threads from one another's behavior, guaranteeing that no thread can halt the progress of other threads. The programmer can thus write a module as a standard computer program, execute it with its own thread, and not have to worry about any other modules in the system. Even though several programs are sharing the processors, programmers can consider each module independently and can think of each module as having a processor to itself. Furthermore, if a programming error causes a module to enter into an endless loop, another module that interacts

with the user gets a chance to run at some point, thus allowing the user to destroy the ill-behaving thread by calling the THREAD_DESTROY procedure.

5.5.5 Enforcing Modularity with Threads and Address Spaces

Preemptive scheduling enforces modularity in the sense that one thread cannot stop the progress of another thread, but if all threads share a single address space, then they can modify each other's memory accidentally. That may be okay for threads that are working together on a common problem, but unrelated threads need to be protected from erroneous or malicious stores of one another. We can provide that protection by making the thread manager aware of the virtual address spaces of Section 5.4.

This awareness can be implemented by having the thread manager, when it switches a processor from one thread to another, also switch the address space. That is, ENTER_PROCESSOR_LAYER saves the contents of the processor's PMAR in the *thread_table* entry of the thread that is releasing the processor, and EXIT_PROCESSOR_LAYER loads the processor's PMAR with the value in the *thread_table* entry of the new thread.

Loading the PMAR adds one significant complication to the thread manager: starting at the instant that the processor loads a new value into the PMAR, the processor will translate virtual addresses using the new page table, so that it will take its next instruction from some location in the new virtual address space. As mentioned in Section 5.4.3.2, one way to deal with this complication is to map both the instructions and the data of the thread manager into the same set of virtual addresses in every virtual address space. Another possibility is to design hardware that can load the PMAR, SP, and PC as a single before-or-after action, thereby returning control to the thread in the new virtual address space at the saved location and with the saved stack pointer.

5.5.6 Layering Threads

Figure 5.23 and the program fragments in Figures 5.24 and 5.25 showed how to create several threads in the thread layer from one thread in the processor layer. In particular, Figure 5.25 explained how a processor thread in the processor layer can be used to dynamically create and delete threads in the thread layer. Many systems generalize this implementation to support interrupt handling and multiple layers of thread management, as shown in Figure 5.27.

To support interrupts, we can think of a processor as a hard-wired thread manager with two threads: (1) a processor thread (e.g., the thread that runs SCHEDULER in Figure 5.25) and (2) an interrupt thread that runs interrupt handlers in kernel mode. On an interrupt, a processor's hard-wired thread manager switches from a processor thread to an interrupt thread that runs an interrupt handler in kernel mode, which may invoke a thread-layer exception handler that calls YIELD.

The operating system thread layer uses the processor threads of the processor layer to implement a second layer of threads and gives each application module one

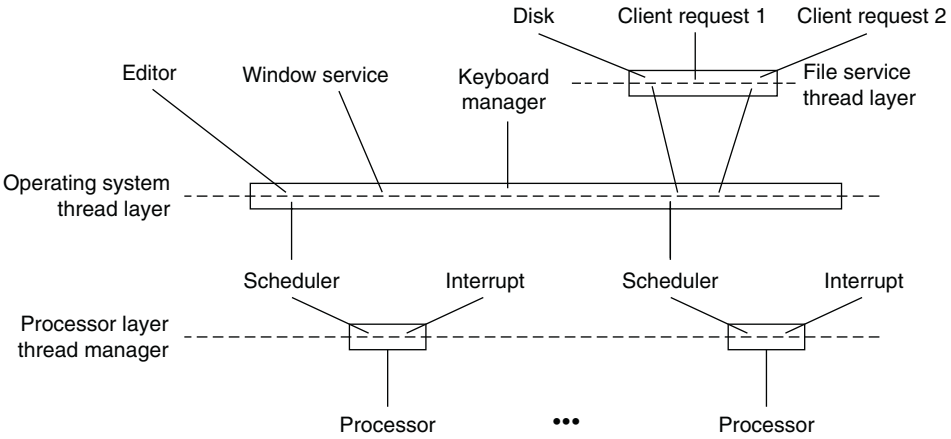


FIGURE 5.27

Thread managers applied recursively.

or more preemptively scheduled virtual processors. When the operating system thread manager switches to another thread, it may also have to load the chosen thread's page-map address into the page-map address register to switch to the address space of the chosen thread. The operating system thread manager runs in kernel mode.

Each application module in turn may implement, if it desires, its own, user-mode, third-layer thread manager using one or more virtual processors provided by the operating system layer. For example, some Web servers have an embedded Java interpreter to run Java programs, which may use several Java threads. To support threads at the Java level, the Java interpreter has its own thread manager. Typically, a third-layer thread manager uses non-preemptive scheduling because all threads belong to the same application module and don't have to be protected from each other.

Generalizing, we get the picture in Figure 5.27, where a number of threads at layer n can be used to implement higher-layer threads at layer $n + 1$. Each hardware processor at the lowest layer creates two threads: a processor thread and an interrupt thread. One layer up, the operating system uses the processor threads to provide one or more threads per module: one thread for the editor, one thread for the window manager, one thread for the keyboard manager, and several threads for the file service. One layer further up, the file service thread creates three application-level threads out of two operating system threads: one to wait for the disk and one for each of two client requests. At each layer, a thread manager switches one or more threads of layer $n - 1$ among several layer n threads.

Although the layering idea is simple in the abstract, in practice a number of issues must be carefully thought through—for example, if a thread blocks in a layer different than the layer where it was created and where its scheduler runs. Clark [Suggestions for Further Reading 5.3.3] and Anderson et al. [Suggestions for Further Reading 5.3.2] discuss some of the practical issues.

5.6 THREAD PRIMITIVES FOR SEQUENCE COORDINATION

The thread manager described in Section 5.5 allows processors to be shared among many threads. A thread can release its processor so that other threads get a chance to run, as the sender and receiver do using YIELD in Figure 5.22. When the sender or receiver is scheduled again, it retests the shared variables *in* and *out*. This mode of interaction, where a thread continually tests a shared variable, is called *polling*. Polling in software is usually undesirable because every time a thread discovers that the test for a shared variable fails, it has acquired and released its processor needlessly. If a system has many polling threads, then the thread manager spends much time performing unnecessary thread switches instead of running threads that have productive work to perform.

Ideally, a thread manager should schedule a thread only when the thread has useful work to perform. That is, we would prefer a way of waiting that avoids spinning on calls to YIELD. For example, a sender with a bounded buffer should be able to tell the thread manager not to run it until $in - out < N$. (That is, until the buffer has room.) One way to approach this goal is for a thread manager to support primitives for sequence coordination, which is what this section explores.

5.6.1 The Lost Notification Problem

To see what we need for the primitives for sequence coordination, consider an obvious, but incorrect, implementation of sender and receiver, as shown in Figure 5.28. This implementation uses a variable shared between the sender and receiver, and two new, but inadequate, primitives—WAIT and NOTIFY—that take as argument the name of the shared variable:

- WAIT (*event_name*) is a before-or-after action that sets this thread's state to WAITING, places *event_name* in the thread table entry for this thread, and yields its processor.
- NOTIFY (*event_name*) is a before-or-after action that looks in the thread table for a thread that is in the state WAITING for *event_name* and changes that thread to the RUNNABLE state.

To support this interface, the thread manager must add the WAITING state to the RUNNING and RUNNABLE state for threads in the thread table. When the scheduler runs (for example, when some thread invokes YIELD), it skips over any thread that is in the WAITING state.

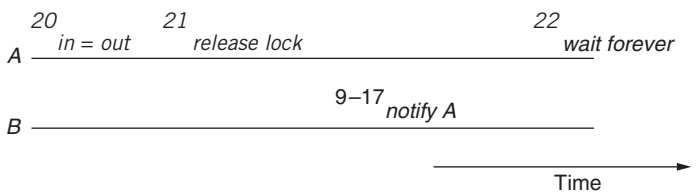
The program in the figure uses these primitives as follows. A thread invokes WAIT to allow the thread manager to release the thread's processor until a call to NOTIFY (lines *15* and *25*). The thread that changes *in* invokes NOTIFY (line *15*) to tell the thread manager to give a processor to a receiver thread waiting on *nonempty* (line *22*), since now there is a message in the buffer (i.e., $out < in$). There is a similar call to NOTIFY by the thread that updates *out* (line *25*), to tell the thread manager to give a processor to a sending thread waiting on *room* (line *12*), since now there is room to add a message to the buffer. This implementation avoids needless thread switches because the waiting receiver thread receives a processor only if NOTIFY has been called.

```
1   shared structure buffer                  // A shared bounded buffer
2       message instance message[N]          // with a maximum of N messages
3       long integer in initially 0          // Counts number of messages put in the buffer
4       long integer out initially 0         // Counts number of messages taken out of the buffer
5       lock instance buffer_lock initially UNLOCKED // Lock to coordinate sender and receiver
6       event instance room                  // Event variable to wait until there is room in buffer
7       event instance notempty              // Event variable to wait until the buffer is not empty

8   procedure SEND (buffer reference p, message instance msg)
9       ACQUIRE (p.buffer_lock)
10      while p.in − p.out = N do            // Wait until there room in the buffer
11          RELEASE (p.buffer_lock)          // Release lock so that receiver can remove
12          WAIT(p.room)                     // Release processor
13          ACQUIRE (p.buffer_lock)
14      p.message[p.in modulo N] ← msg       // Put message in the buffer
15      if p.in = p.out then NOTIFY(p.notempty) // Signal thread that there is a message
16      p.in ← p.in + 1                      // Increment in
17      RELEASE (p.buffer_lock)

18  procedure RECEIVE (buffer reference p)
19      ACQUIRE (p.buffer_lock)
20      while p.in = p.out do                // Wait until there is a message to receive
21          RELEASE (p.buffer_lock)          // Release lock so that sender can add
22          WAIT(p.notempty)                 // Release processor
23          ACQUIRE (p.buffer_lock)
24      msg ← p.message[p.out modulo N]      // Copy item out of buffer
25      if p.in − p.out = N then NOTIFY(p.room) // Signal thread that there is room now
26      p.out ← p.out + 1                    // Increment out
27      RELEASE (p.buffer_lock)
28      return msg
```

FIGURE 5.28

An implementation of a virtual communication link for a system with locks, NOTIFY, and WAIT.

Unfortunately, the use of WAIT and NOTIFY introduces a race condition. Let's assume that the buffer is empty (i.e., *in* = *out*) and a receiver and a sender run on separate processors. The following order of statements will result in a lost notification: *A20*, *A21*, *B9* through *B17*, and *A22*:



The receiver finds that *buffer* is empty (*A20*) and releases the lock (*A21*), but before the receiver executes *A22*, the sender executes *B9* through *B17*, which adds an item

to the buffer and notifies the receiver. The notification is lost because the receiver hasn't called WAIT yet. Now the receiver executes WAIT (*A22*) and waits for a notification that will never come. The sender continues adding items to the buffer until the buffer is full and then calls WAIT. Now both the receiver and sender are waiting.

We could modify the program to call NOTIFY on each invocation of SEND, but that won't fix the problem. It will make it more unlikely that the notification will be lost, but it won't eliminate the possibility. The following ordering of statements could happen: the receiver executes *A20* and *A21*, then it is interrupted long enough for the sender to add N items, and then the receiver calls *A22*. With this ordering, the receiver misses all of the repeated notifications.

Swapping statements *21* and *22* will result in a lost notification too. Then the receiver would call WAIT while still holding the lock on *buffer_lock*. But the sender needs to be able to acquire *buffer_lock* in order to notify the receiver, so everything would come to a halt.

The problem is that we have three operations on the shared buffer state that must be turned into a before-or-after action: (1) testing if there is room in the shared buffer, and (2) if not, going to sleep until there is room and (3) releasing the shared lock so that another thread can make room. If these three operations are not a before-or-after action, then the risk of the lost notification problem arises.

The pseudocode that uses WAIT and NOTIFY illustrates a tension between modularity and locks. An observant reader might ask: if the problem is a race condition caused by having concurrent threads running multistep actions (e.g., the sender (1) tests for space and (2) calls WAIT, at the same time that the receiver (1) makes space and (2) calls NOTIFY), why don't we just make those steps into before-or-after actions by putting a lock around them? The problem is that the steps we would like to make into an atomic action are for the example of the sender (1) comparing *in* and *out* and (2) changing the thread table entry from RUNNING to WAITING. But the variables *in* and *out* are owned by the sender and receiver modules, whereas the thread table is owned by the thread manager module. These are not only separate modules, but the thread manager is probably in the kernel. So who should own the lock that creates the before-or-after action? We can't allow correctness of the kernel to depend on a user program properly setting and releasing a kernel lock, nor can we allow the correctness of the kernel to depend on a user lock being correctly implemented. The real problem here is that the lock needed to create the before-or-after action must protect an invariant that is a relation between a piece of application-owned state and a piece of system-owned state.

5.6.2 Avoiding the Lost Notification Problem with Eventcounts and Sequencers

Designers have identified various solutions to the problem of creating before-or-after actions to eliminate lost notifications. A general property of all these solutions is that they bring some additional thread state that characterizes the event for which the thread is waiting under protection of the thread table lock (i.e., *thread_table_lock*). By extending the semantics of WAIT and NOTIFY to include examining and modifying the

variable *event_name*, it is possible to avoid lost notifications. We leave that solution as an exercise to the reader and instead offer simpler and more widely used solutions based on primitives other than WAIT and NOTIFY. Problem set *13* introduces a solution in which the additional thread state is held in what is called a *condition variable*, and Birrell's tutorial does a nice job of explaining how to program with threads and condition variables [Suggestions for Further Reading 5.3.1]. Sidebar 5.7 and problem set *12* describe a solution in which the additional thread state is a variable known as a *semaphore*. In this section we describe a solution (one that is intended to be particularly easy to reason about) in which the additional thread state is found in variables called *eventcounts* and *sequencers* [Suggestions for Further Reading 5.5.4]. In all of these solutions, the additional thread state must be shared between the application (e.g., SEND and RECEIVE) and the thread manager, so the semantics of WAIT/NOTIFY, condition variables, semaphores, eventcounts, and other similar solutions all contain non-obvious and sometimes quite subtle aspects. A good discussion of some of these subtle issues is provided by Lampson and Redell [Suggestions for Further Reading 5.5.5].

Eventcounts and sequencers are variables that are shared among the sender, the receiver, and the thread manager. They are manipulated using the following interface:

- AWAIT (*eventcount, value*) is a before-or-after action that compares eventcount to value. If *eventcount* exceeds *value*, AWAIT returns to its caller. If *eventcount* is less than or equal to *value*, AWAIT changes the state of the calling thread to WAITING, places *value* and the name of *eventcount* in this thread's entry in the thread table, and yields its processor.

- ADVANCE (*eventcount*) is a before-or-after action that increments *eventcount* by one and then searches the thread table for threads that are waiting on this eventcount. For each one it finds, if *eventcount* now exceeds the value for which that thread is waiting, ADVANCE changes that thread's state to RUNNABLE.

- TICKET (*sequencer*) is a before-or-after action that returns a non-negative value that increases by one on each call. Two threads concurrently calling TICKET on the same *sequencer* receive different values, and the ordering of the values returned corresponds to the time ordering of the execution of TICKET.

- READ (*eventcount* or *sequencer*) is a before-or-after action that returns to the caller the current value of the variable. Having an explicit READ procedure ensures before-or-after atomicity for eventcounts and sequencers whose value may grow to be larger than a memory cell.

To implement this interface, the scheduler skips over any thread that is in the WAITING state.

To understand these primitives, consider first the implementation of a bounded buffer for a single sender and receiver. Using eventcounts, we can rewrite the implementation of the bounded buffer from Figure 5.6 as shown in Figure 5.29. SEND waits until there is space in the buffer. Because AWAIT implements the waiting operation, the code in Figure 5.29 does not need the **while** loop that waits for success in Figure 5.6. Once there is space, the sender adds the message to the buffer and increments *in* using

Sidebar 5.7 Avoiding the Lost Notification Problem with Semaphores Semaphores are counters with special semantics for sequence coordination. A semaphore supports two operations:

- DOWN (*semaphore*): if *semaphore* > 0 decrement *semaphore* and return otherwise, wait until another thread increases *semaphore* and then try to decrement again.

- UP (*semaphore*): increment *semaphore*, wake up all threads waiting on *semaphore*, and return.

Semaphores are inspired by the ones that the railroad system uses to coordinate the use of a shared track. If a semaphore is down, trains must stop until the current train on the track leaves the track and raises the semaphore. If a semaphore can take on only the values 0 and 1 (sometimes called a binary semaphore), then UP and DOWN operate similar to a railroad semaphore. Semaphores were introduced in computer systems by the Dutch programmer Edgar Dijkstra (see also Sidebar 5.2), who called the DOWN operation P ("pakken", for grabbing in Dutch) and the UP operation V ("verhogen", for raising in Dutch) [Suggestions for Further Reading 5.5.1].

The implementation of DOWN and UP must be before-or-after actions to avoid the lost notification problem. This property can be realized in the same way as the eventcount operations:

```
1     structure semaphore
2         integer count
3
4     procedure UP (semaphore reference sem)
5         ACQUIRE (thread_table_lock)
6         sem.count ← sem.count + 1
7         for i from 0 to 6 do  // wakeup all threads waiting on this semaphore
8             if thread_table[i].state = WAITING and thread_table[i].sem = sem then
9                 thread_table[i].state ← RUNNABLE
10        RELEASE (thread_table_lock)

11    procedure DOWN (semaphore reference sem)
12        ACQUIRE (thread_table_lock)
13        id ← GET_THREAD_ID()
14        thread_table[id].sem ← sem  // record the semaphore ID is waiting on
15        while sem.count < 1 do      // give up the processor when sem < 1
16            thread_table[id].state ← WAITING
17            ENTER_PROCESSOR_LAYER (id, CPUID)
18        sem.count ← sem.count − 1
19        RELEASE (thread_table_lock)
```

Using semaphores, one can implement SEND and RECEIVE with a bounded buffer without lost notifications (see problem set *12*).

```
1     shared structure buffer
2         message instance message[N]
3         eventcount instance in initially 0
4         eventcount instance out initially 0

5     procedure SEND (buffer reference p, message instance msg)
6         AWAIT (p.out, p.in − N)                  // Wait until there is space in buffer
7         p.message[READ(p.in) modulo N] ← msg      // Copy message into buffer
8         ADVANCE (p.in)                           // Increment in and alert receiver

9     procedure RECEIVE (buffer reference p)
10        AWAIT (p.in, p.out)                      // Wait till something in buffer
11        msg ← p.message[READ(p.out) modulo N]    // Copy message out of buffer
12        ADVANCE (p.out)                          // Increment out and Alert sender
13        return msg
```

FIGURE 5.29

An implementation of a virtual communication link for a single sender and receiver using eventcounts.

ADVANCE, which may change the receiver's state to RUNNABLE. Because AWAIT and ADVANCE operations are before-or-after actions, the lost notification problem cannot occur.

Again, because AWAIT implements the waiting operation, the receiver implementation is also simple. RECEIVE waits until there is a message in the buffer. If so, the receiver extracts the message from the buffer and increments *out* using ADVANCE, which may change the sender's state to RUNNABLE.

Figure 5.30 shows the implementation for the case of multiple senders with a single receiver. To ensure that several senders don't try to write into the same location within the buffer, we need to coordinate their actions. We can use the TICKET primitive to solve this problem, which requires changes only to SEND. The main difference between Figure 5.30 and Figure 5.29 is that the senders must obtain a ticket to serialize their operations. SEND obtains a ticket from the sequencer *sender* (line *7*). TICKET operates likes the "take a number" machine in a bakery or post office. The returned tickets create an ordering of senders and tell each sender its position in the order. Each sender thread then waits until its turn comes up by invoking AWAIT, passing as arguments the eventcount *sent* and the value of its TICKET (*t*) (line *8*). When *sent* reaches the number on the ticket of a sender thread, that sender thread proceeds to the next step, which is to wait until there is space in the buffer (line *9*), and only then does it add its item to its entry in *buffer*. Because TICKET is a before-or-after action, no two threads will get the same number. Again, because AWAIT and ADVANCE operations are before-or-after actions, the lost notification problem cannot happen.

Again, this solution doesn't use a **while** loop that waits for the success in Figure 5.6. With multiple senders, it is slightly tricky to see why this is correct. AWAIT guarantees that *eventcount* exceeded *value* at some instant after AWAIT was called, but if there are other, concurrent, threads that may increment *value*, by the time AWAIT's caller gets control back, *eventcount* may no longer exceed *value*. The proper view is that a

```
1     shared structure buffer
2         message instance message[N]
3         eventcount instance in initially 0
4         eventcount instance out initially 0
5         sequencer instance sender

6     procedure SEND (buffer reference p, message instance msg)
7         t ← TICKET (p.sender)                    // Allocate a buffer slot
8         AWAIT (p.in, t)                          // Wait till previous slots are filled
9         AWAIT (p.out, READ(p.in) − N)            // Wait till there is space in buffer
10        p.message[READ(p.in) modulo N] ← msg      // Copy message into buffer
11        ADVANCE (p.in)                           // Increment in and alert receiver
```

FIGURE 5.30

An implementation of a virtual communication link for several senders using eventcounts.

return from AWAIT is a hint that the condition AWAIT was waiting for was true and it may still be true, but the program that called AWAIT must check again to be sure.

The issue seems to arise when there are multiple senders. Suppose the buffer is full (say *in* and *out* are 10) and there are two sending threads that are both waiting for a slot to become empty. The first one of those sending threads that runs will absorb the buffer entry and change *in* to 11. The second sending thread will find that *in* is 11 but *out* is also 11, so from its point of view, AWAIT returned with *in* = *out*. Yet it doesn't recheck the condition. Closer inspection of the code reveals that this case can never arise because the second sender is actually waiting its turn on the ticket returned by the sequencer *sender*, not waiting for *in* < *out*. There is never a case in which two senders are waiting for the same condition to become true. If the program had used a different way of coordinating the senders, it might have required a retest of the condition when AWAIT returns. This is another example of why programming with concurrent threads requires great care.

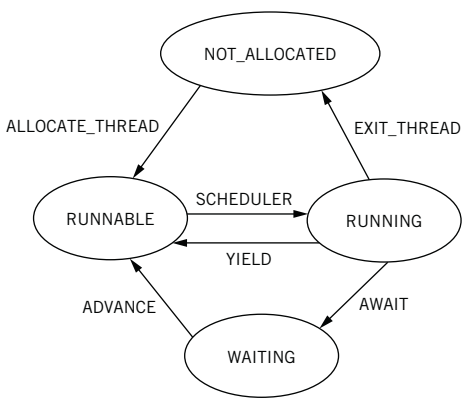


FIGURE 5.31

Thread state diagram. In any of the three states RUNNABLE, WAITING, or RUNNING, a call to DESTROY_THREAD sets a flag that causes the scheduler to force the state to NOT_ALLOCATED the next time that thread would have entered the RUNNING state.

If the implementation must also work with multiple receivers, then a similar sequencer is needed in RECEIVE to allow the receivers to serialize themselves.

With these additional primitives for sequence coordination, we can describe the life of a thread as a state machine with four states (see Figure 5.31). The thread manager

creates a thread in the RUNNABLE state. The thread manager schedules one of the runnable threads and dispatches a processor to it; that thread changes to the RUNNING state. By calling YIELD, the thread reenters the RUNNABLE state, and the manager can select another thread and dispatch to it. Alternatively, a thread can change from the RUNNING state to the NOT_ALLOCATED state by calling EXIT_THREAD. Or a running thread can enter the WAITING state by calling AWAIT when it cannot proceed until some event occurs. Another thread, by calling ADVANCE, can make the waiting thread enter the RUNNABLE state again.

These primitives provide new opportunities for a programmer to create deadlocks. For example, thread A may call AWAIT on an eventcount that it expects thread B to ADVANCE, but thread B may be AWAITing an eventcount that only thread A is in a position to ADVANCE. Eventcounts and tickets can eliminate lost notifications, but the primitives that manipulate them must still be used with care. The last few questions of problem set *11* explore the problem of lost notifications by comparing a simple Web service implemented using NOTIFY and ADVANCE.

5.6.3 Implementing AWAIT, ADVANCE, TICKET, and READ (Advanced Topic)

To implement AWAIT, ADVANCE, TICKET, and READ we extend the thread manager as follows. YIELD doesn't require any modifications to support AWAIT and ADVANCE, but we must extend the *thread_table* to record, for threads in the WAITING state, a reference to the eventcount on which it is waiting:

```
shared structure thread_table[7]
    integer topstack                    // value of the stack pointer
    integer state                       // WAITING, RUNNABLE, TERMINATE, NOT_ALLOCATED
    eventcount reference event          // if waiting, the eventcount we are waiting on
    long integer value                  // if waiting, what value are we waiting for
shared lock instance thread_table_lock  // lock to protect entries of thread_table
```

The field *event* is a reference to an eventcount so that the thread manager and the calling thread can share it. This sharing is the key to resolving the tension mentioned earlier: it allows a calling thread variable to be protected by the thread manager lock.

We implement AWAIT by testing the eventcount, setting the state to WAITING if the test fails, and calling ENTER_PROCESSOR_LAYER to switch to the processor thread:

```
1  structure eventcount
2      long integer count

3  procedure AWAIT (eventcount reference event, value)
4      ACQUIRE (thread_table_lock)
5      id ← GET_THREAD_ID ()
6      thread_table[id].event ← event
7      thread_table[id].value ← value
8      if event.count ≤ value then thread_table[id].state ← WAITING
9      ENTER_PROCESSOR_LAYER (id, CPUID)
10     RELEASE (thread_table_lock)
```

This implementation of AWAIT releases its processor unless eventcount *event* exceeds value in a before-or-after action. As before, the thread data structures are protected by

the lock *thread_table_lock*. In particular, the lock ensures that the line *8* comparison of *event* with *value* and the potential change of state from RUNNING to WAITING are two steps of a before-or-after action that must occur either completely before or completely after any concurrent call to ADVANCE that might change the value of *event* or the state of this thread. The lock thus prevents lost notifications.

ENTER_PROCESSOR_LAYER in AWAIT causes control to switch from this thread to the processor thread, which may give the processor away. The thread that calls ENTER_PROCESSOR_LAYER passes the lock it acquired to the processor thread, which passes it to the next thread to run on this processor. Thus, no other thread can modify the thread state while the thread that invoked AWAIT holds *thread_table_lock*. A return from that call to ENTER_PROCESSOR_LAYER means that some other thread called AWAIT or YIELD and the processor thread has decided it is appropriate to assign a processor to this thread again. The thread will return to line *10*, release *thread_table_lock*, and return to the caller of AWAIT.

The ADVANCE procedure increments the eventcount *event*, finds all threads that are waiting on *count* and whose *value* is less than *count's*, and changes their state to RUNNABLE:

```
1  procedure ADVANCE (eventcount reference event)
2      ACQUIRE (thread_table_lock)
3      event.count ← event.count + 1
4      for i from 0 until 7 do
5          if thread_table[i].state = WAITING and thread_table[i].event = event and
6                                          event.count > thread_table[i].value then
7              thread_table[i].state ← RUNNABLE
8      RELEASE (thread_table_lock)
```

The key in the implementation of ADVANCE is that it uses *thread_table_lock* to make ADVANCE a before-or-after action. In particular, the line *6* comparison of *event.count* with *thread*[*i*].*value* and the line *7* change of *state* to RUNNABLE of the thread that called AWAIT are now two steps of a before-or-after action. No thread calling AWAIT can interfere with a thread that is in ADVANCE. Similarly, no thread calling ADVANCE can interfere with a thread that is in AWAIT. This setup avoids races between AWAIT and ADVANCE, and thus the lost notification problem.

ADVANCE just makes a thread runnable; it doesn't call ENTER_PROCESSOR_LAYER to release its processor. The runnable thread won't run until some other thread (perhaps the caller of ADVANCE) calls YIELD or AWAIT, or until the scheduler preemptively releases a processor.

We implement a sequencer and the TICKET operation as follows:

```
1  structure sequencer
2      long integer ticket

3  procedure TICKET (sequencer reference s)
4      ACQUIRE (thread_table_lock)
5      t ← s.ticket
6      s.ticket ← t + 1
7      RELEASE (thread_table_lock)
8      return t
```

For completeness, the implementation of READ of an eventcount is as follows:

```
1  procedure READ (eventcount reference event)
2      ACQUIRE (thread_table_lock)
3      e ← event.count
4      RELEASE (thread_table_lock)
5      return e
```

To ensure that READ provides before-or-after atomicity, READ is implemented as a before-or-after action using locks. The implementation of READ of a sequencer is similar.

Recall that in Figure 5.8, ACQUIRE itself is implemented with a spin loop, polling the lock continuously instead of releasing the processor. Given that ACQUIRE and RELEASE are used to protect only short sequences of instructions, a spinning implementation is acceptable. Furthermore, inside the thread manager we must use a spinning lock because if ACQUIRE (*thread_table_lock*) were to call AWAIT to wait until the lock is unlocked, then the thread manager would be calling itself, but it isn't designed to be recursive. In particular, it does not have a base case that could stop recursion.

5.6.4 Polling, Interrupts, and Sequence Coordination

Some threads must interact with external devices. For example, the keyboard manager must be able to interact with the keyboard controller on the keyboard, which is a separate, special-purpose processor. As we shall see, this interaction is just another example of sequence coordination.

The keyboard controller is a special-purpose processor, which runs a single program that gathers key strokes. In the terminology of this chapter, we can think of the keyboard controller as a single, hard-wired thread running with its own dedicated processor. When the user presses a key, the keyboard controller thread raises a signal line long enough to set a flip-flop, a digital circuit that can store one bit that the keyboard manager can read. The controller thread then lowers the signal line until next time (i.e., until the next keystroke). The flip-flop shared between the controller and the manager allows them to coordinate their activities.

In fact, using the shared flip-flop, the keyboard manager can run a wait-for-input loop similar to the one in the receiver:

```
1  while FLIP_FLOP = 0 do
2      YIELD ()
```

In this case, the keyboard controller sets the flip-flop and the keyboard manager reads the flip-flip and tests it. If the flip-flop is not set, it reads 0, and the manager yields. If it is set, it falls out of the loop. As a side-effect of reading the flip-flop, it is reset to 0, thus providing a kind of coordination lock.

Here we have another example of polling. In polling, a thread keeps checking whether another (perhaps hardware) thread needs attention. In our example, the keyboard manager runs every time the scheduler offers it a chance, to see if

any new keys have been pressed. The keyboard manager thread is continually in a RUNNABLE state, and whenever the scheduler selects it to run, the thread checks the flip-flop.

Polling has several disadvantages, especially if it is done by a program. If it is difficult to predict the time until the event will occur, then there is no good choice for how often a thread should poll. If the polling thread executes infrequently (e.g., because the processors are busy executing other threads), then it might take a long time before a device receives attention. In this case, the computer system might appear to be unresponsive; for example, if a user must wait a long time before the computer processes the user's keyboard input, the user has a bad interactive experience. On the other hand, if the scheduler selects the polling thread frequently (e.g., faster than users can type), the thread wastes processor cycles, since often there will be no input available. Finally, some devices might require that a processor executes their managers by a certain deadline because otherwise the device won't operate correctly. For example, the keyboard controller may have only a single keystroke register available to communicate with the keyboard manager. If the user types a second keystroke before the keyboard manager gets a chance to run and absorb the first one, the first keystroke may be lost.

These disadvantages are similar to the disadvantages of not having explicit primitives for sequence coordination. Without AWAIT and ADVANCE, the thread scheduler doesn't know when the receiver thread must run; therefore, the receiver thread may make unnecessary, repeated calls to YIELD. This situation with the keyboard manager is similar; ideally, when the controller has input that needs to be processed, it should be able to alert the scheduler that the keyboard manager thread should run. We would like to program the keyboard manager and keyboard controller as a sender and a receiver using the primitives for sequence coordination, much as in Figure 5.30, except we could use a solution that works for a single sender and a single receiver. Unfortunately, the controller cannot invoke procedures such as AWAIT and ADVANCE directly; it shares only a single flip-flop with the processors.

The trick is to move the polling loop down into the hardware by using interrupts. The keyboard manager enables interrupts by setting a processor's interrupt control register to ON, indicating to that processor that it must take interrupts from the keyboard controller. Then, to check for an interrupt, the processor polls the shared flip-flop at the beginning of every instruction cycle. When the processor finds that the shared flip-flop has changed, instead of proceeding to the next instruction, the processor executes the interrupt handler. In other words, interrupts are actually implemented as a polling loop inside a processor. A processor may support multiple interrupts by providing multiple shared flip-flops and a map that associates a different interrupt handler with each flip-flop.

A simple interrupt handler for the keyboard device calls ADVANCE, the call that the keyboard controller is unable make directly, and then returns. The interrupted thread continues operation without realizing that anything happened. But the next time any thread calls YIELD or AWAIT, the thread manager can notice that the keyboard manager thread has become runnable. When it runs, the keyboard manager can then copy the

keystrokes from the device, translate them to a character representation, put them in a shared buffer, (e.g., for the receiver), and wait for the next keystroke.

Because the interrupt handler gains control of a processor within one instruction time, it can be used to meet deadlines. For example, the interrupt handler for the keyboard device could copy the user's keystrokes to a buffer owned by the keyboard manager immediately, instead of waiting until the keyboard manager gets a chance to run. This way the keyboard device is immediately ready for the user's next keystroke. To meet such deadlines, interrupt handlers are usually more elaborate than a single call to ADVANCE. It is common to place modest-sized chunks of code in an interrupt handler to move data out of the device buffers (e.g., keystrokes out of the keyboard device) or immediately restart an I/O device that has turned itself off.

Putting code in an interrupt handler must be done with great care. An interrupt handler must be cautious in reading or writing shared variables because it may be invoked between any pair of instructions. Therefore, the handler cannot be sure of the state of the thread currently running on the processor or on other processors.

Since interrupt handlers are not threads managed by the operating system thread manager, the interrupt handlers and the operating system thread manager must be carefully programmed. For example, the thread manager should call ACQUIRE and RELEASE on the *thread_table_lock* with interrupts disabled because otherwise a deadlock might occur, as we saw in Section 5.5.4. As another example, an interrupt handler should never call AWAIT because AWAIT may release its processor to the surprise of the interrupted thread—the interrupted thread may be a thread that has nothing to do with the interrupt but just happened to be running on the processor when the interrupt occurred. On the other hand, an interrupt handler can invoke ADVANCE without causing any problems.

The restrictions on exception handlers that process errors caused by the currently running thread (e.g., a divide-by-zero error) are less severe because the handler runs on behalf of the thread currently running on the processor. So, in that case, the handler can call YIELD or AWAIT.

5.7 CASE STUDY: EVOLUTION OF ENFORCED MODULARITY IN THE INTEL X86

The previous sections introduced the main ideas for enforcing modularity within a computer using a simple processor. This section presents a case study of how the popular Intel x86 processor provides support for enforced modularity and how commonly used operating systems use this support. The next section provides a case study of enforcing modularity at the processor level using virtual machines.

The Intel x86 processor architecture is currently the most widely used architecture for microprocessors of personal computers, laptops, and servers. The x86 architecture started without any support for enforced modularity. As the robustness of software on personal computers, laptops, and servers has become

important, the Intel designers have added support for enforcing modularity. The Intel designers didn't get it right on the first try. The evolution of x86 architecture to include enforced modularity provides some good examples of the rapid improvement in technology and challenges of designing complex systems, including market pressure.

5.7.1 The Early Designs: No Support for Enforced Modularity

In 1971 Intel produced its first microprocessor, the 4004, intended for calculators and implemented in 2,250 transistors. The 4004 is a 4-bit processor (i.e., the word size is 4 bits and the processor computes with 4-bit wide operands) and can address as much as 4 kilobytes of program memory and 640 bytes of data memory. The 4004 provides a stack that can store only three stack frames, no interrupts, and no support for enforcing modularity. Hardware support for the missing features was well known in 1971, but there is little need for them in a calculator.

The follow-on processor, the 8080 (1974), was Intel's first microprocessor that was used in a personal computer, namely, the Altair, produced by MITS. Unlike the 4004, the 8080 is a general-purpose microprocessor. The 8080 has 5,000 transistors: an 8-bit processor that can address up to 64 kilobytes of memory (16-bit addresses), without support for enforcing modularity. Bill Gates and Paul Allen of Microsoft fame developed a program that could run BASIC applications on the Altair. Since the Altair couldn't run more than a single, simple program at one time, there was still no need for enforcing modularity.

The 8080 was followed by the 8086 in 1978, with 29,000 transistors. The 8086 is a 16-bit processor but with 20-bit bus addresses, allowing access to 1 megabyte of memory. To make a 20-bit address out of a 16-bit address register, the 8086 has four 16-bit wide segment descriptors. The 8086 combines the value in the segment descriptors and the 16-bit address in an operand as follows: (16-bit segment descriptor $\times$ 16) + 16-bit address, producing a 20-bit value. The segment descriptor can be viewed as a memory address to which the 16-bit address in the operand field of the instruction is added.

The primary purpose of these segments is to extend physical memory, as opposed to providing enforced modularity. Using the four segment descriptors, a program can refer to a total of 256 kilobytes of memory at one time. If a program needs to address other memory, the programmer must save one of the segment descriptors and load it with a new value. Thus, writing programs for the 8086 that use more than 256 kilobytes of memory is inconvenient because the programmer must keep track of segment descriptors and where segment data is located.

Although the 8086 has a different instruction repertoire from the 8080, programs for the 8080 could run on the 8086 unmodified using a translator provided by Intel. As we will see, backwards compatibility is a recurring theme in the evolution of the Intel processor architecture and one key to Intel's success.

The 8088 (1979) was put together hastily in response to IBM's request for a processor for its personal computer. The 8088 is identical to the 8086, except that it

has an 8-bit data bus, which made the processor less expensive. Most devices at that time had an 8-bit interface anyway. Microsoft supplied the operating system, named Microsoft Disk Operating System (MS-DOS), for the IBM PC. Microsoft first licensed the operating system from Seattle Computer Products and then acquired it shortly before the release of the PC for $50,000. The IBM PC was a commercial success and started the rise of Intel and Microsoft.

The IBM PC reserved the first 640 kilobytes of the 1-megabyte physical address space for programs and the top 360 kilobytes for input and output. The designers assumed that no programs on a personal computer needed more than 640 kilobytes of memory. To keep the price and complexity down, neither 8088 nor MS-DOS had any support for enforcing modularity.

5.7.2 Enforcing Modularity Using Segmentation

Because the IBM PC was inexpensive, it became widely used; more and more new software was developed for it, and the existing software became richer in features. In addition, users wanted to run several programs at the same time; that is, they wanted to easily switch from one program to another without having to exit a program and start it again later. These developments posed three new design goals for Intel and Microsoft: larger address spaces to run more complex programs, running several programs at once, and enforcing modularity between them. Unfortunately, the last goal conflicts with backwards compatibility because existing programs took full advantage of having direct access to physical memory.

Intel's first attempt to achieve some of these goals was the 80286* (1982), a 16-bit processor that can address up to 16 megabytes of memory (24-bit physical addresses) and has 134,000 transistors. The 80286 has two modes, named *real* and *protected*: in real mode old 8086 programs can run; in protected mode new programs can take advantage of enforced modularity through a change in the interpretation of segment descriptors. In protected mode the segment descriptors don't define the base address of a segment (as in real mode); rather, they select a segment descriptor out of a table of segment descriptors. This application of the design principle ***decouple modules with indirection*** allows a protected-mode program to refer to 2^{14} segments. Furthermore, the low 2 bits of a segment selector are reserved for permission bits; 2 bits supports four protection levels so that operating systems designers can exploit several protection rings†. In practice, protection rings are of limited usefulness, and operating system designers use only two rings (user and

*In 1982 Intel also introduced the 80186 and 80188, but these 6-mHz processors were used mostly as embedded processors instead of processors for personal computing. One of the major contributions of the 80186 is the reduction in the number of chips required because it included a DMA controller, an interrupt controller, and a timer.

†Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM* 15, 3 (March 1972), pages 157–170.

kernel) to ensure, for example, that user-level programs cannot access kernel-only segments.

Although Intel sold 15 million 80286s, it achieved the three goals only partially. First, 24 bits was small compared to the 32 bits of address space offered by competing processors. Second, although it is easy to go from real to protected mode, there was no easy way (other than exploiting an unrelated feature in the design of the processor) to switch from protected mode back to real. This restriction meant that an operating system could not easily switch between old and new programs. Third, it took years after the introduction of the 80286 to develop an operating system, OS/2, that could take advantage of the segmentation provided by the 80286. OS/2 was jointly created by Microsoft and IBM, for the purpose of taking advantage of all the protected-mode features of the 80286. But when Microsoft grew concerned about the project, it disowned OS/2, gave it to IBM, and focused instead on Windows 2.0. Most buyers didn't wait for IBM and Microsoft to get their operating system acts together and instead simply treated the 80286-based PC as a faster 8086 PC that could use more memory.

Overlapping with the 80286, Intel invested over 100 person-years in the design of a full-featured segment-based processor architecture known as the i432. This processor was a ground-up design to enforce modularity and support object-oriented programming. The segment-based architecture included direct support for capabilities, a protection technique for access control (see Chapter 11 [on-line]). The resulting implementation was so complex that it didn't fit on a single chip and it ran slower than the 80286. It was eventually abandoned, not because it enforced modularity, but because it was overly complex, slow, and lacked backward compatibility with the x86 processor architectures.

5.7.3 Page-Based Virtual Address Spaces

Under market pressure from Motorola, which was selling a 32-bit processor with support for page-based virtual memory, Intel scratched the i432 and followed the 80286 with the 80386 in 1985. The 80386 has 270,000 transistors and addresses the main shortcomings of the 80286, while still being backwards compatible with it. The 80386 is a 32-bit processor, which can refer to up to 4 gigabytes of memory (32-bit addresses) and supports 32-bit external data and address busses. Compared with the two real and protected modes of the 80286, the 80386 provides an additional mode, called virtual real mode, which allows several real-mode programs to run at the same time in virtual environments fully protected from one another. The 80386 design also allows a single segment to grow to 2^{32} bytes, the maximum size of physical memory. Within a segment, the 80386 designers added support for virtual memory using paging with a separate page table for each segment. Operating system designers can choose to use virtual memory with segments, or pages, or both.

This design allows several old programs to run in virtual real mode, each in its own paged address space. This design also allows old programs to have access to

more memory than on the 80286, without being forced to use multiple segments. Furthermore, because the 80386 segmentation was backwards compatible with the 80286, 80286 programs and the Windows 2.0 successor (Windows 3.0) could use the larger segments without any modification. For these reasons, the 80386 was a big hit immediately, but it took a while until 32-bit operating systems were available. GNU/Linux, a widely-used open-source UNIX-based system came out in 1991, and Microsoft's Windows 3.1 and IBM's OS/2 2.0 in 1992. All of these systems incorporated the enforced modularity ideas, pioneered in the time-sharing systems of the 1960s and 1970s.

5.7.4 Summary: More Evolution

After 1985, the Intel processor architecture was extended with new instructions, but the core instruction repertoire remained the same. The main changes occurred under the hood. Intel and other companies figured out how to implement processors that provide the complex x86 instruction repertoire—some instructions are 1 byte, and others can be up to 17 bytes long, which is why the literature calls the x86 a Complex Instruction Set Computer, or CISC, while still running as fast as processor architectures designed from scratch with a RISC instruction repertoire. This effort has paid off in terms of performance but has required a large number of transistors to achieve it.

Figure 5.32 shows the growth of Intel processors in terms of transistors over the period 1970–2008*. The *y*-axis is on a logarithmic scale, and the straight line suggests that the growth has been approximately exponential. The Pentium was originally designated the 80586, but Intel redesignated the 80586 the "Pentium" in order to secure a trademark. This growth is a nice example of *d(technology)/dt* in action (see Sidebar 1.6).

The growth in software is also large. Figure 5.33 shows the growth of the Linux kernel in terms of lines of code during the period 1991–2008†. In this graph, the *y*-axis is on a linear scale. As can be seen, the growth in terms of lines of code has been large, and what is shown is just the kernel. A large contributor to this growth is device drivers for new hardware devices.

The success of the x86 illustrates the importance of a specific instance of the *unyielding foundations rule*: provide backwards compatibility. If one must change an interface, keep the old interface around or simulate the old version of the interface using the new version of the interface, so that clients keep working without modifications. It is typically much less work to develop a simulation layer that provides backwards compatibility than to reimplement all of the clients from scratch.

For processors, backwards compatibility is particularly important because legacy software is a big factor in the success of a processor architecture. The reason is that

*Source: Intel Web page (http://www.intel.com/pressroom/kits/quickreffam.htm).

†The sum of number of lines in all C files (source and include) in a kernel release.

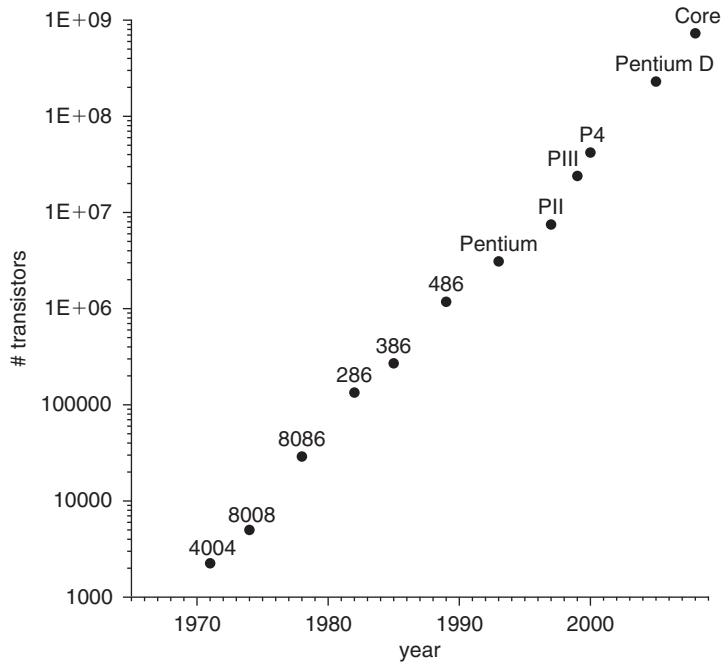


FIGURE 5.32

Growth of the number of transistors in Intel processor chips. The label on each point is the commercial name of the chip. (Log scale on *y*-axis).

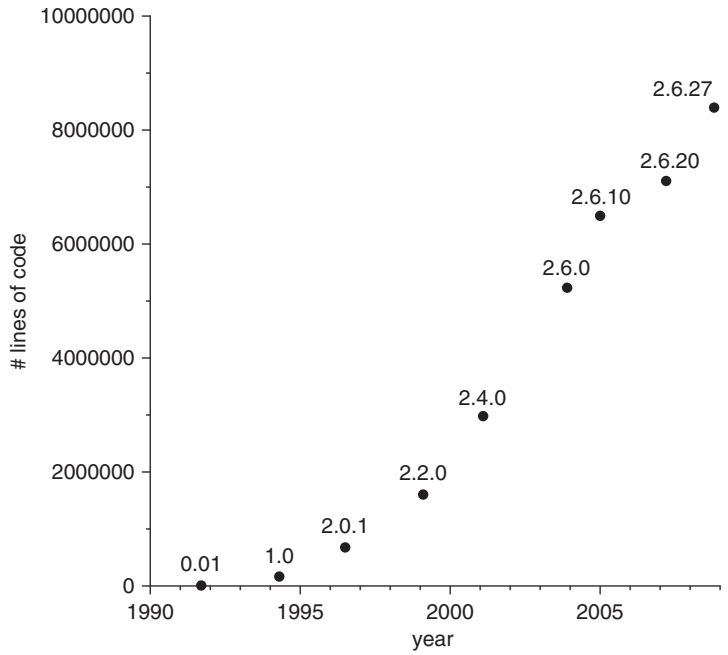


FIGURE 5.33

Growth of the number of lines of code in the Linux kernel. The label on each point is the Linux release number. (Linear scale on *y*-axis).

legacy software is expensive to modify—the original programmers usually have departed (or forgotten about it) and have not documented it well. Experience shows that even minor modifications risk violating some undocumented assumptions, so it is necessary for someone to understand the old program completely, which takes almost as much effort as writing a completely new one. So customers will nearly always choose the architecture that allows them to continue to run legacy software unchanged. Because the x86 architecture provided backwards compatibility, it was able to survive the competition from RISC processors.

Today we see the legacy software scenario being played out in the change from 32-bit virtual addresses to 64-bit virtual addresses. Intel's Itanium architecture is gradually disappearing beneath the waves because it is not backwards compatible, while competitor Advanced Micro Devices (AMD)'s 64-bit Athlon is backwards compatible with the billion or so x86 processors currently in the field. At the time of writing, Intel is abandoning the Itanium architecture and following AMD.

Backwards compatibility can also backfire. For example, Xerox decided it looked more promising to create a PC-clone rather than to commercialize a workstation that Xerox developed in its research lab, which had a mouse, a window manager, and a WYSIWYG editor [Suggestions for Further Reading 1.3.3]. Steve Jobs saw the prototype and developed an equivalent—the Apple Macintosh. The benefits of the Macintosh were so great compared to PCs that customers were willing to buy it. (The later evolution of the Macintosh is a different, less successful story.)

5.8 APPLICATION: ENFORCING MODULARITY USING VIRTUAL MACHINES

This chapter has introduced several high-level abstractions to virtualize processors, memory, and links to enforce modularity. Applications interact with these abstractions through a supervisor-call interface, and interrupt and exception handlers. Another approach uses *virtual machines*. In this approach, a real physical machine is used as much as possible to implement many virtual instances of itself (including its privileged instructions, such as loading and storing to the page-map address register). That is, virtual machines emulate many instances of a machine A using a real machine A. The software that implements the virtual machines is known as a *virtual machine monitor*. This section discusses virtual machines and virtual machine monitors in more detail.

5.8.1 Virtual Machine Uses

A virtual machine is useful in a number of situations:

- To run several *guest* operating systems side by side. For example, on one virtual Intel x86 machine, one can run the GNU/Linux operating system, and on another

one can run the Windows/XP operating system. If the virtual machine monitor implements the Intel x86 faithfully (i.e., instructions, state, protection levels, page tables), then one can run GNU/Linux, Windows/XP, and their applications on top of the monitor without modifications.

- To contain errors in a guest operating system. Because the guest runs inside a virtual machine, errors in the guest operating system cannot affect the operating systems software on other virtual machines. This feature is handy for debugging a new operating system or for containing an operating system that is flaky but important for certain applications.
- To simplify development of operating systems. The virtual machine monitor can virtualize the physical hardware to provide a simpler interface, which may simplify the development of an operating system. For example, the virtual machine monitor may turn a multiprocessor computer into a few uniprocessor computers to allow the guest operating system to be written for a uniprocessor, which simplifies coordination.

5.8.2 Implementing Virtual Machines

Virtual machine monitors can be implemented in two ways. First, one can run the monitor directly on hardware in kernel mode, with the guest operating systems in user mode. Second, one can run the monitor as an application in user mode on top of a *host* operating system. The latter may be less complex to implement because the monitor can take advantage of the abstractions provided by the host operating systems, but it is only possible if the host operating system forwards all the events that monitor needs to perform its job. For simplicity, we assume the first approach (see Figure 5.34); the issues are the same in either case.

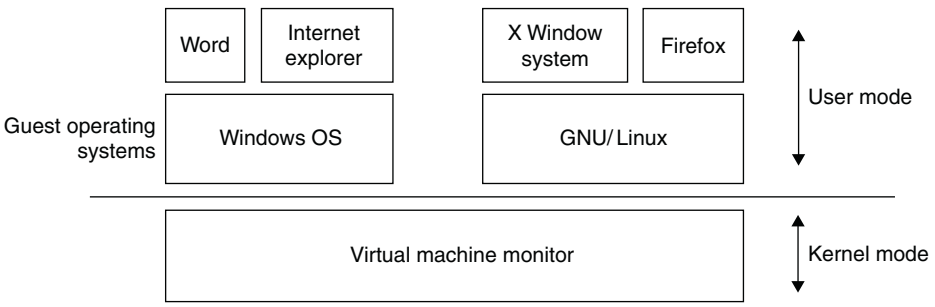


FIGURE 5.34

A virtual machine monitor providing two virtual machines, each running a different guest operating with its own applications.

To implement virtual machine, the virtual machine monitor must provide three primary functions:

1. *Virtualizing the computer*. For example, if a guest operating system stores a new value into the page-map address register, then the monitor must make the guest operating system believe that it can do so, even though the guest is running in user mode.
2. *Dispatch events*. For example, the monitor must forward interrupts, exceptions, and supervisor calls invoked by the applications to the appropriate guest operating systems.
3. *Allocate resources*. For example, the monitor must divide physical memory among the guest operating systems.

Virtualizing the computer is easy if all instructions are *virtualizable*. That is, all the instructions that allow a guest to tell the difference between running on the physical and running on a virtual machine must result in an exception to the monitor so that the monitor can emulate the intended behavior. In addition, the exception must leave enough information for the exception handler to emulate the instruction and restart the guest operating system as if it has executed the instruction.

Consider instructions that load the page-map address register. These instructions behave differently in user mode and kernel mode. In user mode, these instructions result in an illegal instruction exception (because they are privileged), and in kernel mode the hardware performs them. If a guest operating system invokes such an instruction, for example, to switch to another application on the guest, the monitor must emulate that instruction faithfully so that the application will run with the right page-map. Thus, a requirement for such an instruction is that it results in an exception so that the monitor receives control, that it leaves enough information around that the monitor can emulate it, and that the monitor can restart the guest as if it executed the instruction. That is, the guest should not be able to tell that the monitor emulated the instruction.

If an instruction behaves differently in kernel mode than in user mode and doesn't result in an exception, then the instruction is called *non-virtualizable*. For example, on the Intel x86 processor enabling interrupts is done by setting the interrupt-enable bit in a register called `EFLAGS`. This instruction behaves differently in user mode and in kernel mode. In user mode, the instruction does not have any effect (i.e., the processor just ignores it), but in kernel mode, the instruction sets the bit in the `EFLAGS` register and allows interrupts. If a guest operating system invokes this instruction in user space, it will do nothing, but the guest operating system assumes that it is running in kernel mode and that the instruction will enable interrupts. This instruction is an example of a non-virtualizable instruction, and handling instructions like these requires a more sophisticated plan, which is beyond the scope of this text. The paper by Adams and Agesen explains it well [Suggestions for Further Reading 5.6.4].

Allocating resources well among the guest operating systems is more challenging than the usual scheduling problem. For example, the monitor must guess which blocks of physical memory are not in use so that it can use those blocks for other guests; the monitor cannot directly inspect the guest's list of free memory blocks. The paper by Waldspurger introduces a nice trick for addressing this problem [Suggestions for Further Reading 5.6.3]. As another example, the monitor must guess when a guest operating system has no work to do; the monitor cannot directly observe that the guest is in its idle loop. The literature on virtual machines contains schemes to address these challenges.

5.8.3 Virtualizing Example

To make concrete what the implementation challenges of these functions are, consider a guest operating system that implements its own page tables, mapping virtual addresses to physical addresses. Let's assume that this guest operating system runs on the processor developed in this text. The goal of the virtual machine monitor is to run several guest operating system by virtualizing the example processor used in this book (see Section 2.1.2), extended with the instructions documented in this chapter.

To allow each guest operating system to address all physical memory, but not other guests' physical memory, the virtual machine monitor must guard the guest's physical addresses. One way to do so is to virtualize addresses recursively. That is, the guest and virtual machine translate application virtual addresses to virtual machine addresses; the monitor translates machine virtual addresses to physical addresses. One challenge in designing the monitor is to maintain this mapping from application virtual to virtual machine to physical addresses. The general plan is for the monitor to emulate loads and stores to the page-map address register, and keep its own translation map per virtual machine, which we will refer to as the machine map.

The monitor can deduce which virtual machine memory a guest is using and the mappings from virtual to machine addresses when the guest invokes a store instruction to the page-map address register. Because this instruction is privileged, the processor will generate an illegal-instruction exception and transfer control to the monitor. The argument to the store instruction contains the machine address of a page map. The monitor can read that memory and see which virtual machine memory the guest is planning to use and what the guest's mappings from virtual to machine are (including the permissions).

For each machine page (including the one that holds the guest page map), the monitor can allocate a physical page and record in the machine map the translation from virtual to machine to physical address, together with its permissions. Equipped with this information, the monitor can construct a new page map that maps the guest's virtual addresses to physical addresses and install that new map in the real page-map address register (which will succeed since the monitor is

running in kernel mode). Thus, although there are two layers of page maps (virtual to machine and machine to physical), the translation performed by the physical processor is only one level: it translates application virtual addresses directly to physical addresses, using the new page map set up by the monitor. To support this double translation plan efficiently, Intel and AMD have added additional hardware support.

As the final step, the monitor can resume the guest operating system at the instruction after the store to the page-map addresses register, providing the illusion to the guest that it updated the page-map address register directly. Now the guest and the applications can continue execution.

If the guest changes its page map (e.g., it switches to one of its other applications), the monitor will learn about this event because the store to the page-map address register will result in an exception (because the instruction is privileged) and invoke an exception handler in the monitor. The exception handler emulates this instruction by updating the physical page-map address register as above and resumes the guest.

If the monitor wants to switch to another guest OS, it can just switch the page-map address register to the new guest's page map, like a switch between applications.

If the application addresses a page that is not part of its address space, the hardware will generate a missing-page exception, which will invoke an exception handler in the monitor. Then, the exception handler in the monitor can invoke the exception handler of the appropriate guest. The guest exception handler now believes it received the missing-page exception directly from the processor, and it can take appropriate action.

A reader interested in learning more about this topic might find the readings on virtual machines useful [Suggestions for Further Reading 5.6].

EXERCISES

5.1 Chapter 1 discussed four general methods for coping with complexity: modularity, abstraction, hierarchy, and layering.

5.1a Which of those four methods does virtual memory use as its primary organizing scheme?

5.1b Which does a microkernel use? Explain.

1996–1–1c,e

5.2 Alyssa is trying to organize her notes on virtual memory systems, and it occurred to her that virtual memory systems can usefully be analyzed as naming systems. She went through Chapter 3 and made a list of some technical terms about naming systems; that list is on the right, below. She then listed some mechanisms found in virtual memory systems on the left. But she isn't sure which naming

concept goes with which mechanism. Help Alyssa out by telling her which letters on the right apply to each numbered mechanism on the left.

1. page map	**a.** Search path
2. virtual address	**b.** Naming network
3. physical address	**c.** Context reference
4. a TLB entry	**d.** Object
5. the page-map address register	**e.** Name
	f. Context
	g. None of the above

1994–2–4

5.3 The Modest Mini Corporation's best-selling computer allows at most two users to run at a time. Its only addressing architecture feature is a single page map, which creates a simple linear address space for the processor. The time-sharing system for this computer loads the page map with a set of memory block addresses before running a user; to switch to the other user, it reloads the entire page map with a new set of memory block addresses. Normally, the set of memory blocks belonging to one user has no overlap with the set of memory blocks belonging to the other user, except that memory block 19 is always assigned as page 3 in every user's address space, providing a "communication region".

5.3a Protection and privacy are obviously a problem with a completely public communication area, but is there any *other* difficulty in using the communication region for any of the following types of data?

A. The character string name of the payroll file

B. An integer representing the number of names in the current payroll file

C. The virtual memory address, within the communication region, of another data item

D. The virtual memory address of a program that lies outside the communication region

E. A small program that is designed to remain within the communication region and execute there

1980–2–4a

5.3b Ben Bitdiddle has decided that programming with page 3 always preassigned is a nuisance. He has therefore proposed that a call to the system be added that reassigns the communication region to a different page of the calling user's address space, while not affecting the other users. What effect would this proposal have on your answers to 5.3a?

1980–2–4b

5.4 One advantage of a microkernel over a monolithic kernel is that it reduces the load on the translation look-aside buffer, and thereby increases its hit rate and its consequent effect on performance. True or False? Explain.

1994-1-3a

5.5 Louis writes a multithreaded program, which produces an incorrect answer some of the time, but always completes. He suspects a race condition. Which of the following are strategies that can reduce, and with luck eliminate, race conditions in Louis's program?

A. Separate a multithreaded program into multiple single-threaded programs, run each thread in its own address space, and share data between them via a communication link that uses SEND and RECEIVE.

B. Apply the one-writer rule.

C. Ensure that for each shared variable v, it is protected by some lock l_v.

D. Ensure that all locks are acquired in the same order.

2006-1-4

5.6 Which of the following statements about operating system kernels are true?

A. Preemptive scheduling allows the kernel's thread manager to run applications in a way that helps avoid fate sharing.

B. The kernel serves as a trusted intermediary between programs running on the same computer.

C. In an operating system that provides virtual memory, the kernel must be invoked to resolve every memory reference.

D. When a kernel switches a processor from one application to another, the target application sets the page-map address register appropriately after it is running in user space.

2007-1-4

5.7 Two threads, A and B, execute a procedure named GLORP but always at different times (that is, only one of the threads calls the procedure at a given time). GLORP contains the following code:

```
procedure GLORP ()
    ACQUIRE (lock_a)
        ACQUIRE (lock_b)
        ...
        RELEASE (lock_b)
    RELEASE (lock_a)
    ...
    ACQUIRE (lock_b)
        ACQUIRE (lock_a)
        ...
        RELEASE (lock_a)
    RELEASE (lock_b)
```

5.7a Assuming that no other code in other procedures ever acquires more than one lock at a time, can there be a deadlock? (If yes, give an example; if not, argue why not.)

1995-1-3a

5.7b Now, assuming that the two threads can be in the code fragment above at the same time, can the program deadlock? (If yes, give an example; if not, argue why not.)

1995-1-3b

5.8 Consider three threads, concurrently executing the three programs shown here. The variables x, y, and z are integers with initial value 0.

Thread 1:	Thread 2:	Thread 3:
for i **from** 1 **to** 100 **do**	**for** i **from** 1 **to** 100 **do**	**for** i **from** 1 **to** 100 **do**
ACQUIRE (A)	ACQUIRE (B)	ACQUIRE (A)
ACQUIRE (B)	ACQUIRE (C)	ACQUIRE (C)
$x \leftarrow x + 1$	$y \leftarrow z + 1$	$z \leftarrow x + 1$
RELEASE (B)	RELEASE (C)	RELEASE (C)
RELEASE (A)	RELEASE (B)	RELEASE (A)

5.8a Can executing these three threads concurrently produce a deadlock? (If yes, give an example; if not, argue why not.)

1993-1-5a

5.8b Does your answer change if the order of the release operations in each thread is reversed? (If they can deadlock, give an example; if not, argue why not.)

1993-1-5b

Additional exercises relating to Chapter 5 can be found in the problem sets beginning on page 425.