CHAPTER

Elements of Computer System Organization

2

CHAPTER CONTENTS

Principles of Computer System Design: An Introduction

DOI: 10.1016/B978-0-12-374957-4.00011-6

OVERVIEW

Although the number of potential abstractions for computer system components is unlimited, remarkably the vast majority that actually appear in practice fall into one of three well-defined classes: *the memory, the interpreter*, and *the communication link*. These three abstractions are so fundamental that theoreticians compare computer algorithms in terms of the number of data items they must remember, the number of steps their interpreter must execute, and the number of messages they must communicate.

Designers use these three abstractions to organize physical hardware structures, not because they are the only ways to interconnect gates, but rather because

- they supply fundamental functions of recall, processing, and communication,
- so far, these are the only hardware abstractions that have proven both to be widely useful and to have understandably simple interface semantics.

To meet the many requirements of different applications, system designers build layers on this fundamental base, but in doing so they do not routinely create completely different abstractions. Instead, they elaborate the same three abstractions, rearranging and repackaging them to create features that are useful and interfaces that are convenient for each application. Thus, for example, the designer of a general-purpose system such as a personal computer or a network server develops interfaces that exhibit highly refined forms of the same three abstractions. The user, in turn, may see the memory in the form of an organized file or database system, the interpreter in the form of a word processor, a game-playing system, or a high-level programming language, and the communication link in the form of instant messaging or the World Wide Web. On examination, underneath each of these abstractions is a series of layers built on the basic hardware versions of those same abstractions.

A primary method by which the abstract components of a computer system interact is *reference*. What that means is that the usual way for one component to connect to another is by *name*. Names appear in the interfaces of all three of the fundamental abstractions as well as the interfaces of their more elaborate higher-layer counterparts. The memory stores and retrieves objects by name, the interpreter manipulates named objects, and names identify communication links. Names are thus the glue that interconnects the abstractions. Named interconnections can, with proper design, be easy to change. Names also allow the sharing of objects, and they permit finding previously created objects at a later time.

This chapter briefly reviews the architecture and organization of computer systems in the light of abstraction, naming, and layering. Some parts of this review will be familiar to the reader with a background in computer software or hardware, but the systems perspective may provide some new insights into those familiar concepts and

it lays the foundation for coming chapters. Section 2.1 describes the three fundamental abstractions, Section 2.2 presents a model for naming and explains how names are used in computer systems, and Section 2.3 discusses how a designer combines the abstractions, using names and layers, to create a typical computer system, presenting the file system as a concrete example of the use of naming and layering for the memory abstraction. Section 2.4 looks at how the rest of this book will consist of designing some higher-level version of one or more of the three fundamental abstractions, using names for interconnection and built up in layers. Section 2.5 is a case study showing how abstractions, naming, and layering are applied in a real file system.

2.1 THE THREE FUNDAMENTAL ABSTRACTIONS

We begin by examining, for each of the three fundamental abstractions, what the abstraction does, how it does it, its interfaces, and the ways it uses names for interconnection.

2.1.1 Memory

Memory, sometimes called *storage*, is the system component that remembers data values for use in computation. Although memory technology is wide-ranging, as suggested by the list of examples in Figure 2.1, all memory devices fit a simple abstract model that has two operations, named WRITE and READ:

WRITE (*name, value*)
value ← READ (*name*)

The WRITE operation specifies in *value* a value to be remembered and in *name* a name by which one can recall that value in the future. The READ operation specifies in *name* the name of some previously remembered value, and the memory device returns that value. A later call to WRITE that specifies the same name updates the value associated with that name.

Hardware memory devices:
- RAM chip
- Flash memory
- Magnetic tape
- Magnetic disk
- CD-R and DVD-R

Higher level memory systems:
- RAID
- File system
- Database management system

FIGURE 2.1
Some examples of memory devices that may be familiar.

Memories can be either volatile or non-volatile. A *volatile* memory is one whose mechanism of retaining information consumes energy; if its power supply is interrupted for some reason, it forgets its information content. When one turns off the power to a *non-volatile* memory (sometimes called "stable storage"), it retains its content, and when power is again available, READ operations return the same values as before. By connecting a volatile memory to a battery or an

Sidebar 2.1 Terminology: Durability, Stability, and Persistence Both in common English usage and in the professional literature, the terms *durability, stability,* and *persistence* overlap in various ways and are sometimes used almost interchangeably. In this text, we define and use them in a way that emphasizes certain distinctions.

Durability A property of a storage medium: the length of time it remembers.
Stability A property of an object: it is unchanging.
Persistence A property of an active agent: it keeps trying.

Thus, the current chapter suggests that files be placed in a durable storage medium—that is, they should survive system shutdown and remain intact for as long as they are needed. Chapter 8 [on-line] revisits durability specifications and classifies applications according to their durability requirements.

This chapter introduces the concept of stable bindings for names, which, once determined, never again change.

Chapter 7 [on-line] introduces the concept of a persistent sender, a participant in a message exchange who keeps retransmitting a message until it gets confirmation that the message was successfully received, and Chapter 8 [on-line] describes persistent faults, which keep causing a system to fail.

uninterruptible power supply, it can be made *durable*, which means that it is designed to remember things for at least some specified period, known as its *durability*. Even non-volatile memory devices are subject to eventual deterioration, known as *decay*, so they usually also have a specified durability, perhaps measured in years. We will revisit durability in Chapters 8 [on-line] and 10 [on-line], where we will see methods of obtaining different levels of durability. Sidebar 2.1 compares the meaning of durability with two other, related words.

At the physical level, a memory system does not normally name, READ, or WRITE values of arbitrary size. Instead, hardware layer memory devices READ and WRITE contiguous arrays of bits, usually fixed in length, known by various terms such as *bytes* (usually 8 bits, but one sometimes encounters architectures with 6-, 7-, or 9-bit bytes), *words* (a small integer number of bytes, typically 2, 4, or 8), *lines* (several words), and *blocks* (a number of bytes, usually a power of 2, that can measure in the thousands). Whatever the size of the array, the unit of physical layer memory written or read is known as a memory (or storage) *cell*. In most cases, the *name* argument in the READ and WRITE calls is actually the name of a cell. Higher-layer memory systems also READ and WRITE contiguous arrays of bits, but these arrays usually can be of any convenient length, and are called by terms such as *record*, *segment*, or *file*.

2.1.1.1 Read/Write Coherence and Atomicity

Two useful properties for a memory are *read/write coherence* and *before-or-after atomicity*. Read/write coherence means that the result of the READ of a named cell is always the same as the most recent WRITE to that cell. Before-or-after atomicity

means that the result of every READ or WRITE is as if that READ or WRITE occurred either completely before or completely after any other READ or WRITE. Although it might seem that a designer should be able simply to assume these two properties, that assumption is risky and often wrong. There are a surprising number of threats to read/write coherence and before-or-after atomicity:

- *Concurrency.* In systems where different actors can perform READ and WRITE operations concurrently, they may initiate two such operations on the same named cell at about the same time. There needs to be some kind of arbitration that decides which one goes first and to ensure that one operation completes before the other begins.
- *Remote storage.* When the memory device is physically distant, the same concerns arise, but they are amplified by delays, which make the question of "which WRITE was most recent?" problematic and by additional forms of failure introduced by communication links. Section 4.5 introduces remote storage, and Chapter 10 [on-line] explores solutions to before-or-after atomicity and read/write coherence problems that arise with remote storage systems.
- *Performance enhancements.* Optimizing compilers and high-performance processors may rearrange the order of memory operations, possibly changing the very meaning of "the most recent WRITE to that cell" and thereby destroying read/write coherence for concurrent READ and WRITE operations. For example, a compiler might delay the WRITE operation implied by an assignment statement until the register holding the value to be written is needed for some other purpose. If someone else performs a READ of that variable, they may receive an old value. Some programming languages and high-performance processor architectures provide special programming directives to allow a programmer to restore read/write coherence on a case-by-case basis. For example, the Java language has a SYNCHRONIZED declaration that protects a block of code from read/write incoherence, and Hewlett-Packard's Alpha processor architecture (among others) includes a *memory barrier* (MB) instruction that forces all preceding READs and WRITEs to complete before going on to the next instruction. Unfortunately, both of these constructs create opportunities for programmers to make subtle mistakes.
- *Cell size incommensurate with value size.* A large value may occupy multiple memory cells, in which case before-or-after atomicity requires special attention. The problem is that both reading and writing of a multiple-cell value is usually done one cell at a time. A reader running concurrently with a writer that is updating the same multiple-cell value may end up with a mixed bag of cells, only some of which have been updated. Computer architects call this hazard *write tearing.* Failures that occur in the middle of writing multiple-cell values can further complicate the situation. To restore before-or-after atomicity, concurrent readers and writers must somehow be coordinated, and a failure in the middle of an update must leave either all or none of the intended update intact. When these conditions are met, the READ or WRITE is said to be *atomic.* A closely related

risk arises when a small value shares a memory cell with other small values. The risk is that if two writers concurrently update different values that share the same cell, one may overwrite the other's update. Atomicity can also solve this problem. Chapter 5 begins the study of atomicity by exploring methods of coordinating concurrent activities. Chapter 9 [on-line] expands the study of atomicity to also encompass failures.

- *Replicated storage.* As Chapter 8 [on-line] will explore in detail, reliability of storage can be increased by making multiple copies of values and placing those copies in distinct storage cells. Storage may also be replicated for increased performance, so that several readers can operate concurrently. But replication increases the number of ways in which concurrent READ and WRITE operations can interact and possibly lose either read/write coherence or before-or-after atomicity. During the time it takes a writer to update several replicas, readers of an updated replica can get different answers from readers of a replica that the writer hasn't gotten to yet. Chapter 10 [on-line] discusses techniques to ensure read/write coherence and before-or-after atomicity for replicated storage.

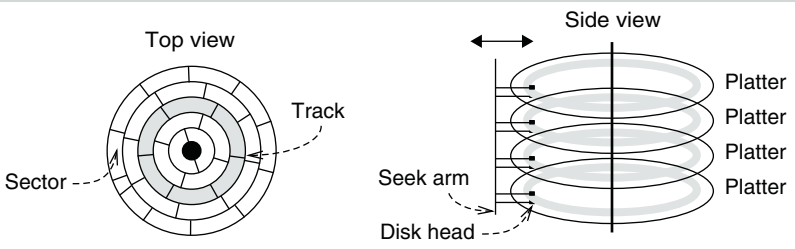
Often, the designer of a system must cope with not just one but several of these threats simultaneously. The combination of replication and remoteness is particularly challenging. It can be surprisingly difficult to design memories that are both efficient and also read/write coherent and atomic. To simplify the design or achieve higher performance, designers sometimes build memory systems that have weaker coherence specifications. For example, a multiple processor system might specify: "The result of a READ will be the value of the latest WRITE if that WRITE was performed by the same processor." There is an entire literature of "data consistency models" that explores the detailed properties of different memory coherence specifications. In a layered memory system, it is essential that the designer of a layer know precisely the coherence and atomicity specifications of any lower layer memory that it uses. In turn, if the layer being designed provides memory for higher layers, the designer must specify precisely these two properties that higher layers can expect and depend on. Unless otherwise mentioned, we will assume that physical memory devices provide read/write coherence for individual cells, but that before-or-after atomicity for multicell values (for example, files) is separately provided by the layer that implements them.

2.1.1.2 Memory Latency

An important property of a memory is the time it takes for a READ or a WRITE to complete, which is known as its *latency* (often called *access time*, though that term has a more precise definition that will be explained in Sidebar 6.4). In the magnetic disk memory (described in Sidebar 2.2) the latency of a particular sector depends on the mechanical state of the device at the instant the user requests access. Having read a sector, one may measure the time required to also read a different but nearby sector in microseconds—but only if the user anticipates the second read and requests it before the disk rotates past that second sector. A request just a few microseconds late may encounter

Sidebar 2.2 How Magnetic Disks Work Magnetic disks consist of rotating circular platters coated on both sides with a magnetic material such as ferric oxide. An electromagnet called a *disk head* records information by aligning the magnetic field of the particles in a small region on the platter's surface. The same disk head reads the data by sensing the polarity of the aligned particles as the platter spins by. The disk spins continuously at a constant rate, and the disk head actually floats just a few nanometers above the disk surface on an air cushion created by the rotation of the platter.

From a single position above a platter, a disk head can read or write a set of bits, called a *track*, located a constant distance from the center. In the top view below, the shaded region identifies a track. Tracks are formatted into equal-sized blocks, called *sectors*, by writing separation marks periodically around the track. Because all sectors are the same size, the outer tracks have more sectors than the inner ones.



A typical modern disk module, known as a "hard drive" because its platters are made of a rigid material, contains several platters spinning on a common axis called a *spindle*, as in the side view above. One disk head per platter surface is mounted on a comb-like structure that moves the heads in unison across the platters. Movement to a specific track is called *seeking*, and the comb-like structure is known as a *seek arm*. The set of tracks that can be read or written when the seek arm is in one position (for example, the shaded regions of the side view) is called a *cylinder*. Tracks, platters, and sectors are each numbered. A sector is thus addressed by geometric coordinates: track number, platter number, and rotational position. Modern disk controllers typically do the geometric mapping internally and present their clients with an address space consisting of consecutively numbered sectors.

To read or write a particular sector, the disk controller first seeks the desired track. Once the seek arm is in position, the controller waits for the beginning of the desired sector to rotate under the disk head, and then it activates the head on the desired platter. Physically encoding digital data in analog magnetic domains usually requires that the controller write complete sectors.

The time required for disk access is called *latency*, a term defined more precisely in Chapter 6. Moving a seek arm takes time. Vendors quote seek times of 5 to 10 milliseconds, but that is an average over all possible seek arm moves. A move from one

(*Sidebar continues*)

cylinder to the next may require only 1/20 of the time of a move from the innermost to the outermost track. It also takes time for a particular sector to rotate under the disk head. A typical disk rotation rate is 7200 rpm, for which the platter rotates once in 8.3 milliseconds. The time to transfer the data depends on the magnetic recording density, the rotation rate, the cylinder number (outer cylinders may transfer at higher rates), and the number of bits read or written. A platter that holds 40 gigabytes transfers data at rates between 300 and 600 megabits per second; thus a 1-kilobyte sector transfers in a microsecond or two. Seek time and rotation delay are limited by mechanical engineering considerations and tend to improve only slowly, but magnetic recording density depends on materials technology, which has improved both steadily and rapidly for many years.

Early disk systems stored between 20 and 80 megabytes. In the 1970s Kenneth Haughton, an IBM inventor, described a new technique of placing disk platters in a sealed enclosure to avoid contamination. The initial implementation stored 30 megabytes on each of two spindles, in a configuration known as a 30–30 drive. Haughton nicknamed it the "Winchester", after the Winchester 30–30 rifle. The code name stuck, and for many years hard drives were known as Winchester drives. Over the years, Winchester drives have gotten physically smaller while simultaneously evolving to larger capacities.

a delay that is a thousand times longer, waiting for that second sector to again rotate under the read head. Thus the maximum rate at which one can transfer data to or from a disk is dramatically larger than the rate one would achieve when choosing sectors at random. A *random access memory (RAM)* is one for which the latency for memory cells chosen at random is approximately the same as the latency for cells chosen in the pattern best suited for that memory device. An electronic memory chip is usually configured for random access. Memory devices that involve mechanical movement, such as optical disks (CDs and DVDs) and magnetic tapes and disks, are not.

For devices that do not provide random access, it is usually a good idea, having paid the cost in delay of moving the mechanical components into position, to READ or WRITE a large block of data. Large-block READ and WRITE operations are sometimes relabeled GET and PUT, respectively, and this book uses that convention. Traditionally, the unqualified term *memory* meant random-access volatile memory and the term *storage* was used for non-volatile memory that is read and written in large blocks with GET and PUT. In practice, there are enough exceptions to this naming rule that the words "memory" and "storage" have become almost interchangeable.

2.1.1.3 Memory Names and Addresses

Physical implementations of memory devices nearly always name a memory cell by the geometric coordinates of its physical storage location. Thus, for example, an electronic memory chip is organized as a two-dimensional array of flip-flops, each holding one named bit. The access mechanism splits the bit name into two parts, which in

turn go to a pair of multiplexers. One multiplexer selects an x-coordinate, the other a y-coordinate, and the two coordinates in turn select the particular flip-flop that holds that bit. Similarly, in a magnetic disk memory, one component of the name electrically selects one of the recording platters, while a distinct component of the name selects the position of the seek arm, thereby choosing a specific track on that platter. A third name component selects a particular sector on that track, which may be identified by counting sectors as they pass under the read head, starting from an index mark that identifies the first sector.

It is easy to design hardware that maps geometric coordinates to and from sets of names consisting of consecutive integers (0, 1, 2, etc.). These consecutive integer names are called *addresses*, and they form the *address space* of the memory device. A memory system that uses names that are sets of consecutive integers is called a *location-addressed memory*. Because the addresses are consecutive, the size of the memory cell that is named does not have to be the same as the size of the cell that is read or written. In some memory architectures each byte has a distinct address, but reads and writes can (and in some cases must always) occur in larger units, such as a word or a line.

For most applications, consecutive integers are not exactly the names that one would choose for recalling data. One would usually prefer to be allowed to choose less constrained names. A memory system that accepts unconstrained names is called an *associative memory*. Since physical memories are generally location-addressed, a designer creates an associative memory by interposing an associativity layer, which may be implemented either with hardware or software, that maps unconstrained higher-level names to the constrained integer names of an underlying location-addressed memory, as in Figure 2.2. Examples of software associative memories, constructed on top of one or more underlying location-addressed memories, include personal telephone directories, file systems, and corporate database systems. A *cache*, a device that remembers the result of an expensive computation in the hope of not redoing that computation if it is needed again soon, is sometimes implemented as an

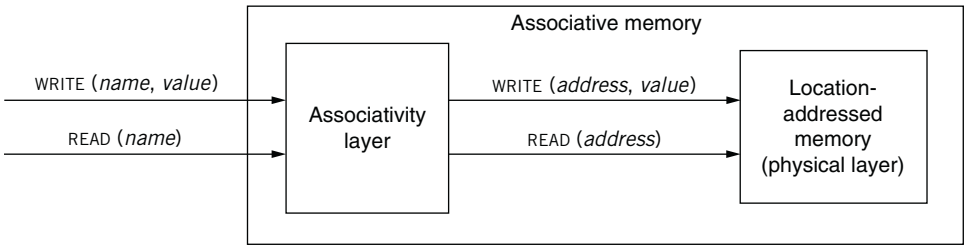


FIGURE 2.2

An associative memory implemented in two layers. The associativity layer maps the unconstrained names of its arguments to the consecutive integer addresses required by the physical layer location-addressed memory.

associative memory, either in software or hardware. (The design of caches is discussed in Section 6.2.)

Layers that provide associativity and name mapping figure strongly in the design of all memory and storage systems. For example, Table 2.2 on page 93 lists the layers of the UNIX file system. For another example of layering of memory abstractions, Chapter 5 explains how memory can be virtualized by adding a name-mapping layer.

2.1.1.4 Exploiting the Memory Abstraction: RAID

Returning to the subject of abstraction, a system known as RAID provides an illustration of the power of modularity and of how the storage abstraction can be applied to good effect. RAID is an acronym for Redundant Array of Independent (or Inexpensive) Disks. A RAID system consists of a set of disk drives and a controller configured with an electrical and programming interface that is identical to the interface of a single disk drive, as shown in Figure 2.3. The RAID controller intercepts READ and WRITE requests coming across its interface, and it directs them to one or more of the disks. RAID has two distinct goals:

- Improved performance, by reading or writing disks concurrently
- Improved durability, by writing information on more than one disk

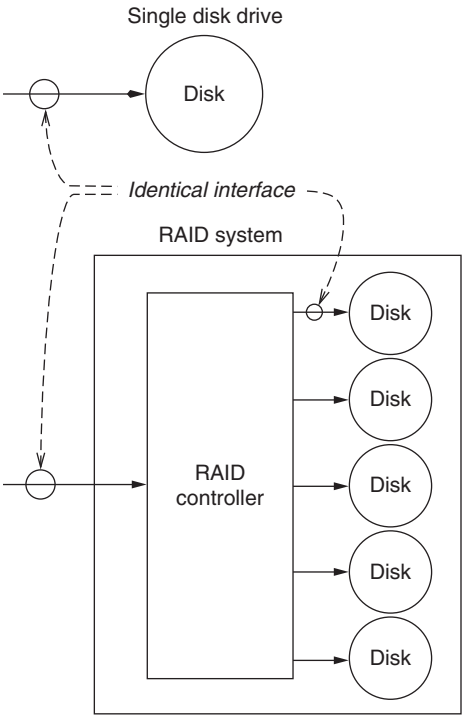


FIGURE 2.3

Abstraction in RAID. The READ/WRITE electrical and programming interface of the RAID system, represented by the solid arrow, is identical to that of a single disk.

Different RAID configurations offer different trade-offs between these goals. Whatever trade-off the designer chooses, because the interface abstraction is that of a single disk, the programmer can take advantage of the improvements in performance and durability without reprogramming.

Certain useful RAID configurations are traditionally identified by (somewhat arbitrary) numbers. In later chapters, we will encounter several of these numbered configurations. The configuration known as RAID 0 (in Section 6.1.5) provides increased performance by allowing concurrent reading and writing. The configuration known as RAID 4 (shown in Figure 8.6 [on-line]) improves disk reliability by applying error-correction codes. Yet another configuration known as RAID 1 (in Section 8.5.4.6 [on-line]) provides high durability by

making identical copies of the data on different disks. Exercise 8.8 [on-line] explores a simple but elegant performance optimization known as RAID 5. These and several other RAID configurations were originally described in depth in a paper by Randy Katz, Garth Gibson, and David Patterson, who also assigned the traditional numbers to the different configurations [see Suggestions for Further Reading 10.2.2].

2.1.2 Interpreters

Interpreters are the active elements of a computer system; they perform the *actions* that constitute computations. Figure 2.4 lists some examples of interpreters that may be familiar. As with memory, interpreters also come in a wide range of physical manifestations. However, they too can be described with a simple abstraction, consisting of just three components:

1. An *instruction reference*, which tells the interpreter where to find its next instruction
2. A *repertoire*, which defines the set of actions the interpreter is prepared to perform when it retrieves an instruction from the location named by the instruction reference
3. An *environment reference*, which tells the interpreter where to find its *environment*, the current state on which the interpreter should perform the action of the current instruction

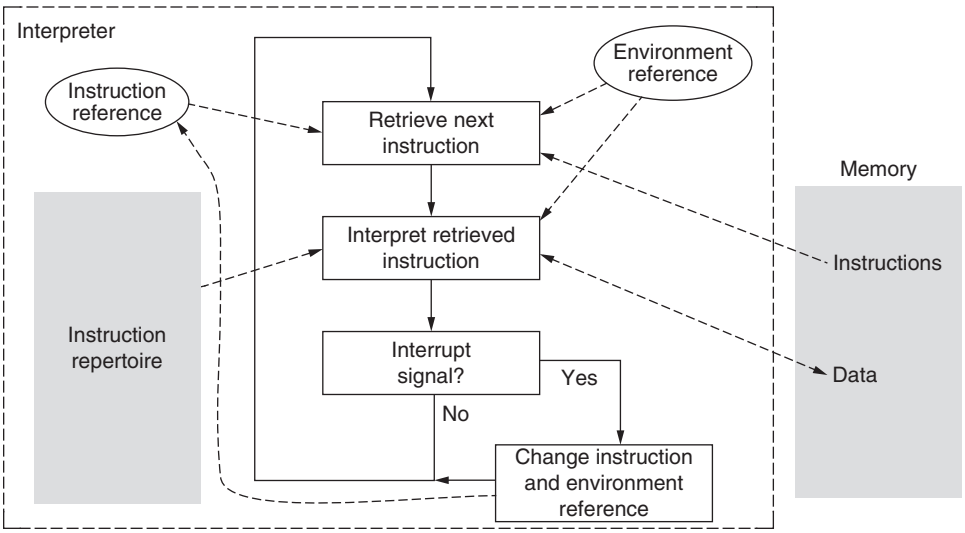
The normal operation of an interpreter is to proceed sequentially through some program, as suggested by the diagram and pseudocode of Figure 2.5. Using the environment reference to find the current environment, the interpreter retrieves from that environment the program instruction indicated in the instruction reference. Again using the environment reference, the interpreter performs the action directed by the program instruction. That action typically involves using and perhaps changing data in the environment, and also an appropriate update of the instruction reference. When it finishes performing the instruction, the interpreter moves on, taking as its next instruction the one now named by the instruction reference. Certain events, called *interrupts*, may catch the attention of the interpreter, causing it, rather than the program, to supply the next instruction. The original program no longer controls the interpreter; instead, a different program, the interrupt handler, takes control and handles the event. The interpreter may also change the environment reference to one that is appropriate for the interrupt handler.

Hardware:
- Pentium 4, PowerPC 970, UltraSPARC T1
- disk controller
- display controller

Software:
- Alice, AppleScript, Perl, Tcl, Scheme
- LISP, Python, Forth, Java bytecode
- JavaScript, Smalltalk
- TeX, LaTeX
- Safari, Internet Explorer, Firefox

FIGURE 2.4

Some common examples of interpreters. The disk controller example is explained in Section 2.3 and the Web browser examples are the subject of Exercise 4.5.




```
1    procedure INTERPRET()
2        do forever
3            instruction ← READ (instruction_reference)
4            perform instruction in the context of environment_reference
5            if interrupt_signal = TRUE then
6                instruction_reference ← entry point of INTERRUPT_HANDLER
7                environment_reference ← environment ref of INTERRUPT_HANDLER
```

FIGURE 2.5

Structure of, and pseudocode for, an abstract interpreter. Solid arrows show control flow, and dashed arrows suggest information flow. Sidebar 2.3 describes this book's conventions for expressing pseudocode.

Sidebar 2.3 Representation: Pseudocode and Messages This book presents many examples of program fragments. Most of them are represented in pseudocode, an imaginary programming language that adopts familiar features from different existing programming languages as needed and that occasionally intersperses English text to characterize some step whose exact detail is unimportant. The pseudocode has some standard features, several of which this brief example shows.

```
1    procedure SUM (a, b)     // Add two numbers.
2        total ← a + b
3        return total
```

The line numbers on the left are not part of the pseudocode; they are there simply to allow the text to refer to lines in the program. Procedures are explicitly declared

(*Sidebar continues*)

(as in line *1*), and indentation groups blocks of statements together. Program variables are set in *italic*, program key words in **bold**, and literals such as the names of procedures and built-in constants in SMALL CAPS. The left arrow denotes substitution or assignment (line *2*) and the symbol "=" denotes equality in conditional expressions. The double slash precedes comments that are not part of the pseudocode. Various forms of iteration (**while**, **until**, **for each**, **do occasionally**), conditionals (**if**), set operations (**is in**), and case statements (**do case**) appear when they are helpful in expressing an example. The construction **for** *j* **from** 0 **to** 3 iterates four times; array indices start at 0 unless otherwise mentioned. The construction *y.x* means the element named *x* in the structure named *y*. To minimize clutter, the pseudocode omits declarations wherever the meaning is reasonably apparent from the context. Procedure parameters are passed by value unless the declaration **reference** appears. Section 2.2.1 of this chapter discusses the distinction between use by value and use by reference. When more than one variable uses the same structure, the declaration *structure_name* **instance** *variable_name* may be used.

The notation *a*(11. . .15) denotes extraction of bits 11 through 15 from the string *a* (or from the variable *a* considered as a string). Bits are numbered left to right starting with zero, with the most significant bit of integers first (using big-endian notation, as described in Sidebar 4.3). The + operator, when applied to strings, concatenates the strings.

Some examples are represented in the instruction repertoire of an imaginary reduced instruction set computer (RISC). Because such programs are cumbersome, they appear only when it is essential to show how software interacts with hardware.

In describing and using communication links, the notation

$$x \Rightarrow y\text{: } \{M\}$$

represents a message with contents *M* from sender *x* to recipient *y*. The notation {*a, b, c*} represents a message that contains the three named fields marshaled in some way that the recipient presumably understands how to unmarshal.

Many systems have more than one interpreter. Multiple interpreters are usually *asynchronous*, which means that they run on separate, uncoordinated, clocks. As a result, they may progress at different rates, even if they are nominally identical and running the same program. In designing algorithms that coordinate the work of multiple interpreters, one usually assumes that there is no fixed relation among their progress rates and therefore that there is no way to predict the relative timing, for example, of the LOAD and STORE instructions that they issue. The assumption of interpreter asynchrony is one of the reasons memory read/write coherence and before-or-after atomicity can be challenging design problems.

2.1.2.1 Processors

A general-purpose processor is an implementation of an interpreter. For purposes of concrete discussion throughout this book, we use a typical reduced instruction set processor. The processor's instruction reference is a *program counter*, stored in a fast memory register inside the processor. The program counter contains the address of the memory location that stores the next instruction of the current program. The environment reference of the processor consists in part of a small amount of built-in location-addressed memory in the form of named (by number) registers for fast access to temporary results of computations.

Our general-purpose processor may be directly wired to a memory, which is also part of its environment. The addresses in the program counter and in instructions are then names in the address space of that memory, so this part of the environment reference is wired in and unchangeable. When we discuss virtualization in Chapter 5, we will extend the processor to refer to memory indirectly via one or more registers. With that change, the environment reference is maintained in those registers, thus allowing addresses issued by the processor to map to different names in the address space of the memory.

The repertoire of our general-purpose processor includes instructions for expressing computations such as adding two numbers (ADD), subtracting one number from another (SUB), comparing two numbers (CMP), and changing the program counter to the address of another instruction (JMP). These instructions operate on values stored in the named registers of the processor, which is why they are colloquially called "op-codes".

The repertoire also includes instructions to move data between processor registers and memory. To distinguish program instructions from memory operations, we use the name LOAD for the instruction that READs a value from a named memory cell into a register of the processor and STORE for the instruction that WRITEs the value from a register into a named memory cell. These instructions take two integer arguments, the name of a memory cell and the name of a processor register.

The general-purpose processor provides a *stack*, a push-down data structure that is stored in memory and used to implement procedure calls. When calling a procedure, the caller pushes arguments of the called procedure (the callee) on the stack. When the callee returns, the caller pops the stack back to its previous size. This implementation of procedures supports recursive calls because every invocation of a procedure always finds its arguments at the top of the stack. We dedicate one register for implementing stack operations efficiently. This register, known as the *stack pointer*, holds the memory address of the top of the stack.

As part of interpreting an instruction, the processor increments the program counter so that, when that instruction is complete, the program counter contains the address of the next instruction of the program. If the instruction being interpreted is a JMP, that instruction loads a new value into the program counter. In both cases, the flow of instruction interpretation is under control of the running program.

The processor also implements interrupts. An interrupt can occur because the processor has detected some problem with the running program (e.g., the program attempted to execute an instruction that the interpreter does not or cannot

implement, such as dividing by zero). An interrupt can also occur because a signal arrives from outside the processor, indicating that some external device needs attention (e.g., the keyboard signals that a key press is available). In the first case, the interrupt mechanism may transfer control to an *exception* handler elsewhere in the program. In the second case, the interrupt handler may do some work and then return control to the original program. We shall return to the subject of interrupts and the distinction between interrupt handlers and exception handlers in the discussion of threads in Chapter 5.

In addition to general-purpose processors, computer systems typically also have special-purpose processors, which have a limited repertoire. For example, a clock chip is a simple, hard-wired interpreter that just counts: at some specified frequency, it executes an `ADD` instruction, which adds 1 to the contents of a register or memory location that corresponds to the clock. All processors, whether general-purpose or specialized, are examples of interpreters. However, they may differ substantially in the repertoire they provide. One must consult the device manufacturer's manual to learn the repertoire.

2.1.2.2 Interpreter Layers

Interpreters are nearly always organized in layers. The lowest layer is usually a hardware engine that has a fairly primitive repertoire of instructions, and successive layers provide an increasingly rich or specialized repertoire. A full-blown application system may involve four or five distinct layers of interpretation. Across any given layer interface, the lower layer presents some repertoire of possible instructions to the upper layer. Figure 2.6 illustrates this model.

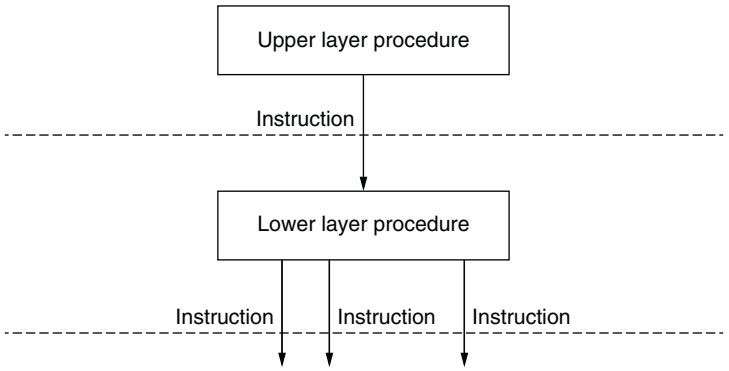


FIGURE 2.6

The model for a layered interpreter. Each layer interface, shown as a dashed line, represents an abstraction barrier, across which an upper layer procedure requests execution of instructions from the repertoire of the lower layer. The lower layer procedure typically implements an instruction by performing several instructions from the repertoire of a next lower layer interface.

Consider, for example, a calendar management program. The person making requests by moving and clicking a mouse views the calendar program as an interpreter of the mouse gestures. The instruction reference tells the interpreter to obtain its next instruction from the keyboard and mouse. The repertoire of instructions is the set of available requests—to add a new event, to insert some descriptive text, to change the hour, or to print a list of the day's events. The environment is a set of files that remembers the calendar from day to day.

The calendar program implements each action requested by the user by invoking statements in some programming language such as Java. These statements—such as iteration statements, conditional statements, substitution statements, procedure calls—constitute the instruction repertoire of the next lower layer. The instruction reference keeps track of which statement is to be executed next, and the environment is the collection of named variables used by the program. (We are assuming here that the Java language program has not been compiled directly to machine language. If a compiler is used, there would be one less layer.)

The actions of the programming language are in turn implemented by hardware machine language instructions of some general-purpose processor, with its own instruction reference, repertoire, and environment reference.

Figure 2.7 illustrates the three layers just described. In practice, the layered structure may be deeper—the calendar program is likely to be organized with an internal upper layer that interprets the graphical gestures and a lower layer that manipulates the calendar data, the Java interpreter may have an intermediate byte-code interpreter layer, and some machine languages are implemented with a microcode interpreter layer on top of a layer of hardware gates.

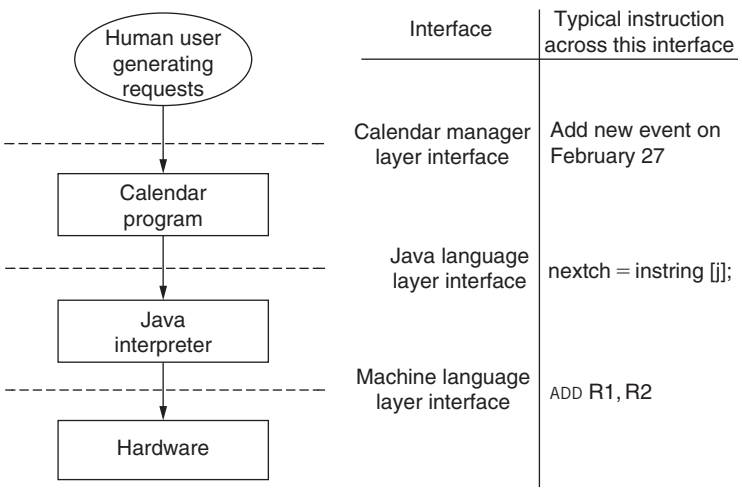


FIGURE 2.7

An application system that has three layers of interpretation, each with its own repertoire of instructions.

One goal in the design of a layered interpreter is to ensure that the designer of each layer can be confident that the layer below either completes each instruction successfully or does nothing at all. Half-finished instructions should never be a concern, even if there is a catastrophic failure. That goal is another example of atomicity, and achieving it is relatively difficult. For the moment, we simply assume that interpreters are atomic, and we defer the discussion of how to achieve atomicity to Chapter 9 [on-line].

2.1.3 Communication Links

A *communication link* provides a way for information to move between physically separated components. Communication links, of which a few examples are listed in Figure 2.8, come in a wide range of technologies, but, like memories and interpreters, they can be described with a simple abstraction. The communication link abstraction has two operations:

SEND (*link_name*, *outgoing_message_buffer*)
RECEIVE (*link_name*, *incoming_message_buffer*)

The SEND operation specifies an array of bits, called a *message,* to be sent over the communication link identified by *link_name* (for example, a wire). The argument *outgoing_message_buffer* identifies the message to be sent, usually by giving the address and size of a buffer in memory that contains the message. The RECEIVE operation accepts an incoming message, again usually by designating the address and size of a buffer in memory to hold the incoming message. Once the lowest layer of a system has received a message, higher layers may acquire the message by calling a RECEIVE interface of the lower layer, or the lower layer may "upcall" to the higher layer, in which case the interface might be better characterized as DELIVER (*incoming_message*).

Names connect systems to communication links in two different ways. First, the *link_name* arguments of SEND and RECEIVE identify one of possibly several available communication links attached to the system. Second, some communication links are actually multiply-attached networks of links, and some additional method is needed to name which of several possible recipients should receive the message. The name of the intended recipient is typically one of the components of the message.

At first glance, it might appear that sending and receiving a message is just an example of copying an array of bits from one memory to another memory over a wire using a sequence of READ and WRITE operations,

Hardware technology:
- twisted pair
- coaxial cable
- optical fiber

Higher level
- Ethernet
- Universal Serial Bus (USB)
- the Internet
- the telephone system
- a UNIX pipe

FIGURE 2.8
Some examples of communication links.

so there is no need for a third abstraction. However, communication links involve more than simple copying—they have many complications, such as a wide range of operating parameters that makes the time to complete a SEND or RECEIVE operation unpredictable, a hostile environment that threatens integrity of the data transfer, asynchronous operation that leads to the arrival of messages whose size and time of delivery can not be known in advance, and most significant, the message may not even be delivered. Because of these complications, the semantics of SEND and RECEIVE are typically quite different from those associated with READ and WRITE. Programs that invoke SEND and RECEIVE must take these different semantics explicitly into account. On the other hand, some communication link implementations do provide a layer that does its best to hide a SEND/RECEIVE interface behind a READ/WRITE interface.

Just as with memory and interpreters, designers organize and implement communication links in layers. Rather than continuing a detailed discussion of communication links here, we defer that discussion to Section 7.2 [on-line], which describes a three-layer model that organizes communication links into systems called *networks*. Figure 7.18 [on-line] illustrates this three-layer network model, which comprises a link layer, a network layer, and an end-to-end layer.

2.2 NAMING IN COMPUTER SYSTEMS

Computer systems use names in many ways in their construction, configuration, and operation. The previous section mentioned memory addresses, processor registers, and link names, and Figure 2.9 lists several additional examples, some of which are probably familiar, others of which will turn up in later chapters. Some system names resemble those of a programming language, whereas others are quite different. When building systems out of subsystems, it is essential to be able to use a subsystem without having to know details of how that subsystem refers to its components. Names are thus used to achieve modularity, and at the same time, modularity must sometimes hide names.

We approach names from an object point of view: the computer system manipulates *objects*. An interpreter performs the manipulation under control of a program or perhaps under the direction of a human user. An object may be structured, which means that it uses other objects as components. In a direct analogy with two ways in which procedures can pass arguments, there are two ways to arrange for one object to use another as a component:

- create a copy of the component object and include the copy in the using object (use by *value*), or
- choose a name for the component object and include just that name in the using object (use by *reference*). The component object is said to *export* the name.

When passing arguments to procedures, use by value enhances modularity, because if the callee accidentally modifies the argument it does not affect the original. But use by value can be problematic because it does not easily permit two or more objects to *share* a component object whose value changes. If both object A

R5	(processor register)
$174FFF_{hex}$	(memory address)
pedantic.edu	(network attachment point name)
18.72.0.151	(network attachment point address)
alice	(user name)
alice@pedantic.edu	(e-mail address)
/u/alice/startup_plan.doc	(file name)
http://pedantic.edu/alice/home.html	(WWW URL)

FIGURE 2.9
Examples of names used in systems.

and B use object C by value, then changing the value of C is a concept that is either meaningless or difficult to implement—it could require tracking down the two copies of C included in A and B to update them. Similarly, in procedure calls it is sometimes useful to give the callee the ability to modify the original object, so most programming languages provide some way to pass the name (pseudocode in this text uses the **reference** declaration for that purpose) rather than the value. One purpose of names, then, is to allow use by reference and thus simplify the sharing of changeable objects.

Sharing illustrates one fundamental purpose for names: as a communication and an organizing tool. Because two uses of the same name can refer to the same object, whether those uses are by different users or by the same user at different times, names are invaluable both for communication and for organization of things so that one can find them later.

A second fundamental purpose for a name is to allow a system designer to defer to a later time an important decision: to which object should this name refer? A name also makes it easy to change that decision later. For example, an application program may refer to a table of data by name. There may be several versions of that table, and the decision about which version to use can wait until the program actually needs the table.

Decoupling one object from another by using a name as an intermediary is known as *indirection*. Deciding on the correspondence between a name and an object is an example of *binding*. Changing a binding is a mechanically easy way to replace one object with another. Modules are objects, so naming is a cornerstone of modularity.

This section introduces a general model for the use of names in computer systems. Some parts of this model should be familiar; the discussion of the three fundamental abstractions in the previous section introduced names and some naming terminology. The model is only one part of the story. Chapter 3 discusses in more depth the many decisions that arise in the design of naming schemes.

2.2.1 The Naming Model

It is helpful to have a model of how names are associated with specific objects. A system designer creates a *naming scheme*, which consists of three elements. The first element is a *name space*, which comprises an alphabet of symbols together with syntax rules that specify which names are acceptable. The second element is

a *name-mapping algorithm*, which associates some (not necessarily all) names of the name space with some (again, not necessarily all) values in a *universe of values*, which is the third and final element of the naming scheme. A *value* may be an object, or it may be another name from either the original name space or from a different name space. A name-to-value mapping is an example of a *binding*, and when such a mapping exists, the name is said to be *bound* to the value. Figure 2.10 illustrates.

In most systems, typically several distinct naming schemes are in operation simultaneously. For example, a system may be using one naming scheme for e-mail mailbox names, a second naming scheme for Internet hosts, a third for files, and a fourth for virtual memory addresses. When a program interpreter encounters a name, it must know which naming scheme to invoke. The environment surrounding use of the name usually provides enough information to identify the naming scheme. For example, in an application program, the author of that program knows that the program should expect file names to be interpreted only by the file system and Internet host names to be interpreted only by some network service.

The interpreter that encounters the name runs the name-mapping algorithm of the appropriate naming scheme. The name-mapping algorithm *resolves* the name, which means that it discovers and returns the associated value (for this reason, the name-mapping algorithm is also called a *resolver*). The name-mapping algorithm is usually controlled by an additional parameter, known as a *context*. For a given naming scheme, there can be many different contexts, and a single name of the name space may map to different values when the resolver uses different contexts. For example, in ordinary discourse when a person refers to the names "you", "here", or "Alice", the meaning of each of those names depends on the context in which the person utters it. On the other hand, some naming schemes have only one context. Such naming schemes provide what are called *universal name spaces*, and they have the nice property that a name always has the same meaning within that naming scheme, no matter who uses it. For example, in the United States, social security numbers, which identify government pension and tax accounts, constitute a universal name space.

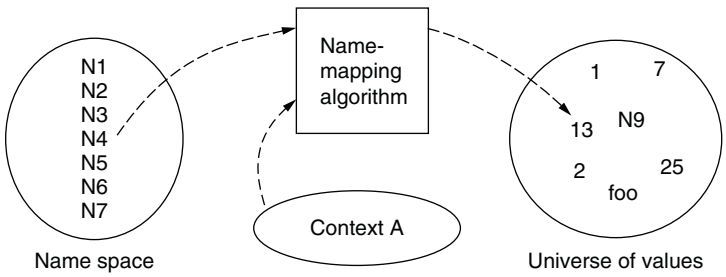


FIGURE 2.10

General model of the operation of a naming scheme. The name-mapping algorithm takes in a name and a context, and it returns an element from the universe of values. The arrows indicate that, using context "A", the algorithm resolves the name "N4" to the value "13".

When there is more than one context, the interpreter may tell the resolver which one it should use or the resolver may use a default context.

We can summarize the naming model by defining the following conceptual operation on names:

value ← RESOLVE (*name*, *context*)

When an interpreter encounters a name in an object, it first figures out what naming scheme is involved and thus which version of RESOLVE it should invoke. It then identifies an appropriate context, resolves the name in that context, and replaces the name with the resolved value as it continues interpretation. The variable *context* tells RESOLVE which context to use. That variable contains a name known as a ***context reference***.

In a processor, register numbers are names. In a simple processor, the set of register names, and the registers those names are bound to, are both fixed at design time. In most other systems that use names (including the register naming scheme of some high-performance processors), it is possible to create new bindings and delete old ones, ***enumerate*** the name space to obtain a list of existing bindings, and compare two names. For these purposes we define four more conceptual operations:

status ← BIND (*name*, *value*, *context*)
status ← UNBIND (*name*, *context*)
list ← ENUMERATE (*context*)
result ← COMPARE (*name1*, *name2*)

The first operation changes *context* by adding a new binding; the *status* result reports whether or not the change succeeded (it might fail if the proposed *name* violates the syntax rules of the name space). After a successful call to BIND, RESOLVE will return the new *value* for *name*.* The second operation, UNBIND, removes an existing binding from *context*, with *status* again reporting success or failure (perhaps because there was no such existing binding). After a successful call to UNBIND, RESOLVE will no longer return that *value* for *name*. The BIND and UNBIND operations allow the use of names to make connections between objects and change those connections later. A designer of an object can, by using a name to refer to a component object, choose the object to which that name is bound either then or at a later time by invoking BIND, and eliminate a binding that is no longer appropriate by invoking UNBIND, all without modifying the object that uses the name. This ability to delay and change bindings is a powerful tool used in the design of nearly all systems. Some naming implementations provide an ENUMERATE operation, which returns a list of all the names that can be resolved in *context*. Some implementations of ENUMERATE can also return a list of all values currently bound in *context*. Finally, the COMPARE operation reports (TRUE or FALSE) whether or not *name1* is the same as *name2*. The meaning of "same" is an interesting question addressed in Section 2.2.5, and it may require supplying additional context arguments.

*The WRITE operation of the memory abstraction creates a name-value association, so it can be viewed as a specialized instance of BIND. Similarly, the READ operation can be viewed as a specialized instance of RESOLVE.

Different naming schemes have different rules about the uniqueness of name-to-value mappings. Some naming schemes have a rule that a name must map to exactly one value in a given context and a value must have only one name, while in other naming schemes one name may map to several values, or one value may have several names, even in the same context. Another kind of uniqueness rule is that of a *unique identifier name space*, which provides a set of names that will never be reused for the lifetime of the name space and, once bound, will always remain bound to the same value. Such a name is said to have a *stable binding.* If a unique identifier name space also has the rule that a value can have only one name, the unique names become useful for keeping track of objects over a long period of time, for comparing references to see if they are to the same object, and for coordination of multiple copies in systems where objects are replicated for performance or reliability. For example, the customer account number of most billing systems constitutes a unique identifier name space. The account number will always refer to the same customer's account as long as that account exists, despite changes in the customer's address, telephone number, or even personal name. If a customer's account is deleted, that customer's account number will not someday be reused for a different customer's account. Named fields within the account, such as the balance due, may change from time to time, but the binding between the customer account number and the account itself is stable.

The name-mapping algorithm plus a single context do not necessarily map all names of the name space to values. Thus, a possible outcome of performing RESOLVE can be a *not-found* result, which RESOLVE may communicate to the caller either as a reserved value or as an exception. On the other hand, if the naming scheme allows one name to map to several values, a possible outcome can be a list of values. In that case, the UNBIND operation may require an additional argument that specifies which value to unbind. Finally, some naming schemes provide *reverse lookup*, which means that a caller can supply a value as an argument to the name-mapping algorithm, and find out what name or names are bound to that value.

Figure 2.10 illustrates the naming model, showing a name space, the corresponding universe of values, a name-mapping algorithm, and a context that controls the name-mapping algorithm.

In practice, one encounters three frequently used name-mapping algorithms:

- Table lookup
- Recursive lookup
- Multiple lookup

The most common implementation of a context is a table of {*name*, *value*} pairs. When the implementation of a context is a table, the name-mapping algorithm is just a lookup of the name in that table. The table itself may be complex, involving hashing or B-trees, but the basic idea is still the same. Binding a new name to a value consists of adding that {*name*, *value*} pair to the table. Figure 2.11 illustrates this common implementation of the naming model. There is one such table for each context, and different contexts may contain different bindings for the same name.

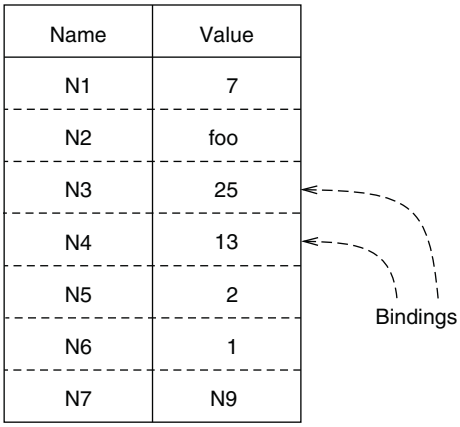


FIGURE 2.11

A system that uses table lookup as the name-mapping algorithm. As in the example of Figure 2.10, this system also resolves the name "N4" to the value "13".

Real-world examples of both the general naming model and the table-lookup implementation abound:

1. A telephone book is a table-lookup context that binds names of people and organizations to telephone numbers. As in the data communication network example, telephone numbers are themselves names that the telephone company resolves into physical line appearances, using a name-mapping algorithm that involves area codes, exchanges, and physical switchgear. The telephone books for Boston and for San Francisco are two contexts of the same naming scheme; any particular name may appear in both telephone books, but if so, it is probably bound to different telephone numbers.
2. Small integers name the registers of a processor. The value is the register itself, and the mapping from name to value is accomplished by wiring.
3. Memory cells are similarly named with the numbers called addresses, and the name-to-value mapping is again accomplished by wiring. Chapter 5 describes an address-renaming mechanism known as virtual memory, which binds blocks of virtual addresses to blocks of contiguous memory cells. When a system implements multiple virtual memories, each virtual memory is a distinct context; a given address can refer to a different memory cell in each virtual memory. Memory cells can also be shared among virtual memories, in which case the same memory cell may have the same (or different) addresses in different virtual memories, as determined by the bindings.
4. A typical computer file system uses several layers of names and contexts: disk sectors, disk partitions, files, and directories are all named objects. Directories are examples of table-lookup contexts. A particular file name may appear in several different directories, bound to either the same or different files. Section 2.5 presents a case study of naming in the UNIX file system.
5. Computers connect to data communication networks at places known as *network attachment points*. Network attachment points are usually named with two distinct naming schemes. The first one, used inside the network, involves a name space consisting of numbers in a fixed-length field. These names are bound, sometimes permanently and sometimes only briefly, to physical entrance

and exit points of the network. A second naming scheme, used by clients of the network, maps a more user-friendly universal name space of character strings to names of the first name space. Section 4.4 is a case study of the Domain Name System, which provides user-friendly attachment point naming for the Internet.

6. A programmer identifies procedure variables by names, and each activation of the procedure provides a distinct context in which most such names are resolved. Some names, identified as "static" or "global names", may instead be resolved in a context that is shared among activations or among different procedures. When a procedure is compiled, some of the original user-friendly names of variables may be replaced with integer identifiers that are more convenient for a machine to manipulate, but the naming model still holds.

7. A Uniform Resource Locator (URL) of the World Wide Web is mapped to a specific Web page by a relatively complicated algorithm that breaks the URL up into several constituent parts and resolves the parts using different naming schemes; the result eventually identifies a particular Web page. Section 3.2 is a case study of this naming scheme.

8. A customer billing system typically maintains at least two kinds of names for each customer account. The account number names the account in a unique identifier name space, but there is also a distinct name space of personal names that can also be used to identify the account. Both of these names are typically mapped to account records by a database system, so that accounts can be retrieved either by account number or by personal name.

These examples also highlight a distinction between "naming" and binding. Some, but not all, contexts "name" things, in the sense that they map a name to an object that is commonly thought of as having that name. Thus, the telephone directory does not "name" either people or telephone lines. Somewhere else there are contexts that bind names to people and that bind telephone numbers to particular physical phones. The telephone directory binds the names of people to the names of telephones.

For each of these examples a context reference must identify the context in which the name-mapping algorithm should resolve the name. Next, we explore where context references come from.

2.2.2 Default and Explicit Context References

When a program interpreter encounters a name in an object, someone must supply a context reference so that the name-mapping algorithm can know which context it should use to resolve the name. Many apparently puzzling problems in naming can be simply diagnosed: the name-mapping algorithm, for whatever reason, used the wrong context reference.

There are two ways to come up with a context with which to resolve the names found in an object: default and explicit. A *default context reference* is one that the resolver supplies, whereas an *explicit context reference* is one that comes packaged

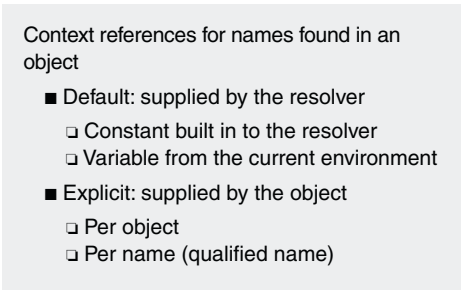


FIGURE 2.12

Taxonomy of context references.

with the name-using object. Sometimes a naming scheme allows for use of both explicit and default methods: it uses an explicit context reference if the object or name provides one; if not, it uses a default context. Figure 2.12 outlines the taxonomy of context references described in the next two paragraphs.

A default context reference can be a constant that is built in to the resolver as part of its design. Since a constant allows for just one context, the resulting name space is universal. Alternatively, a default context reference can be a variable that the resolver obtains from its current execution environment. That variable may be set by some context assignment rule. For example, in most multiple-user systems, each user's execution environment contains a state variable called the *working directory*. The working directory acts as the default context for resolving file names. Similarly, the system may assign a default context for each distinct activity of a user or even, as will be seen in Chapter 3 (Figures 3.2 and 3.3), for each major subsystem of a system.

In contrast, an explicit context reference commonly comes in one of two forms: a single-context reference intended to be used for all the names that an object uses, or a distinct context reference associated with each name in the object. The second form, in which each name is packaged with its own context reference, is known as a *qualified name*.

A context reference is itself a name (it names the context), which leads some writers to describe it as a *base name*. The name resolver must thus resolve the name represented by the context reference before it can proceed with the original name resolution. This recursion may be repeated several times, but it must terminate somewhere, with the invocation of a name resolver that has a single built-in context. This built-in context contains the bindings that permit the recursion to be unraveled.

That description is quite abstract. To make it concrete, let's revisit the previous real-world examples of names, in each case looking for the context reference the resolver uses:

1. When looking up a number in a telephone book, you must provide the context reference: you need to know whether to pick up the Boston or the San Francisco telephone book. If you call Directory Assistance to ask for a number, the operator will immediately ask you for the context reference by saying, "What city, please?" If you got the name from a personal letter, that letter may mention the city—an example of an explicit context reference. If not, you may have to guess, or undertake a search of the directories of several different cities.

2. In a processor, there is usually only one set of numbered registers; they comprise a default context that is built-in using wires. Some processors have multiple

register sets, in which case there is an additional register, usually hidden from the application programmer, that determines which register set is currently in use. The processor uses the contents of that register, which is a component of the current interpretation environment, as a default context reference. It resolves that number with a built-in context that binds register set numbers to physical register sets by interpreting the register set number as an address that locates the correct bank of registers.

3. In a system that implements multiple virtual memories, the interpretation environment includes a processor register (the page-map address register of Chapter 5) that names the currently active page table; that register contains a reference to the default context. Some virtual memory systems provide a feature known as *segments*. In those systems, a program may issue addresses that contain an explicit context reference known as a *segment number*. Segments are discussed in Section 5.4.5.

4. In a file system with many directories, when a program refers to a file using an unqualified or incompletely qualified file name, the file system uses the working directory as a default context reference. Alternatively, a program may use an *absolute path name*, an example of a fully qualified name that we will discuss in depth in just a moment. The path name contains its own explicit context reference. In both the working directory and the absolute path name, the context reference is itself a name that the resolver must resolve before it can proceed with the original name resolution. This need leads to recursive name resolution, which is discussed in Section 2.2.3.

5. In the Internet, names of network attachment points may be qualified (e.g., `ginger.pedantic.edu`) or unqualified (e.g., `ginger`). When the network name resolver encounters an unqualified name, it qualifies that name with a default context reference, sometimes called the default domain. However it materializes, a qualified name is an absolute path name that still needs to be resolved. A different default—usually a configuration parameter of the name resolver—supplies the context for resolution of that absolute path name in the universal name space of Internet domain names. Section 4.4 describes in detail the rather elaborate mechanism that resolves Internet domain names.

6. The programming language community uses its own terminology to describe default and explicit context references. When implementing *dynamic scope*, the resolver uses the current naming environment as a default context for resolving names. When implementing *static* (also called *lexical*) *scope*, the creator of an object (usually a procedure object) associates the object with an explicit context reference—the naming environment at that instant. The language community calls this combination of an object and its context reference a *closure*.

7. For resolution of a URL for the World Wide Web, the name resolver is distributed, and different contexts are used for different components of the URL. Section 3.2 provides details.

8. Database systems provide the contexts for resolution of both account numbers and personal names in a billing system. If the billing system has a graphical user interface, it may offer a lookup form with blank fields for both account number and personal name. A customer service representative chooses the context reference by typing in one of the two fields and hitting a "find" button, which invokes the resolver. Each of the fields corresponds to a different context.

A context reference can be dynamic, meaning that it changes from time to time. An example is when the user clicks on a menu button labeled "Help". Although the button may always appear in the same place on the screen, the context in which the name "Help" is resolved (and thus the particular help screen that appears in response) is likely to depend on which application program, or even which part of that program, is running at the instant that the user clicks on the button.

A common problem is that the object that uses a name does not provide an explicit context, and the name resolver chooses the wrong default context. For example, a file system typically resolves a file name relative to a current working directory, even though this working directory may be unrelated to the identity of the program or data object making the reference. Compared with the name resolution environment of a programming system, most file systems provide a rather primitive name resolution mechanism.

An electronic mail system provides an example of the problem of making sure that names are interpreted in the intended context. Consider the e-mail message of Figure 2.13, which originated at Pedantic University. In this message, `Alice`, `Bob`, and `Dawn` are names from the local name space of e-mailboxes at Pedantic University, and `Charles@cse.Scholarly.edu` is a qualified name of an e-mailbox managed by a mail service named `cse.Scholarly.edu` at the Institute of Scholarly Studies. The name `Charles` is of a particular mailbox at that mail service, and the @-sign is conventionally used to separate the name of the mailbox from the name of the mail service.

As it stands, if user `Charles` tries to reply to the sender of this message, the response will be addressed to `Bob`. Since the first name resolver to encounter the reply message is probably inside the system named `cse.Scholarly.edu`, that resolver would in the normal course of events use as a default context reference the name of the local mail service. That is, it would try to send the message to `Bob@cse.Scholarly.edu`. That isn't the mailbox address of the user who sent the original message. Worse, it might be someone else's mailbox address.

```
To: Bob
Cc: Charles@cse.Scholarly.edu
From: Alice
------
Based on Dawn's suggestions, this chapter has
experienced a major overhaul this year. If you
like it, send your compliments to Dawn (her e-mail
address is "Dawn"); if you do not like it, send your
complaints to me.
```

FIGURE 2.13
An e-mail message that uses default contexts.

```
To: Bob@Pedantic.edu
cc: Charles@cse.Scholarly.edu
From: Alice@Pedantic.edu
------
Based on Dawn's suggestions, this chapter has
experienced a major overhaul this year. If you
like it, send your compliments to Dawn (her e-mail
address is "Dawn"); if you do not like it, send
your complaints to me.
```

FIGURE 2.14

The e-mail message of Figure 2.13 after the mail system expands every unqualified address in the headers to include an explicit context reference.

When constructing the e-mail message, Alice intended local names such as `Bob` to be resolved in her own context. Most mail sending systems know that a local name is not useful to anyone outside the local context, so it is conventional for the mail system to tinker with unqualified names found in the address fields by automatically rewriting them as qualified names, thus adding an explicit context reference to the names `Bob` and `Alice`, as shown in Figure 2.14.

Unfortunately, the mail system can perform this address rewriting only for the headers because that is the only part of the message format it fully understands. If an e-mail address is embedded in the text of the message (as in the example, the mailbox name `Dawn`), the mail system has no way to distinguish it from the other text. If the recipient of the message wishes to make use of an e-mail address found in the text of the message, that recipient is going to have to figure out what context reference is needed. Sometimes it is easy to figure out what to do, but if a message has been forwarded a few times, or the recipient is unaware of the problem, a mistake is likely.

A partial solution could be to tag the e-mail message with an explicit context reference, using an extra header, as in Figure 2.15. With this addition, a recipient of this message could select either `Alice` in the header or `Dawn` in the text and ask the mail system to send a reply. The mail system could, by examining the `Context:` header, determine how to resolve any unqualified e-mail address associated with this message, whether found in the original headers or extracted from the text of the message. This scheme is quite *ad hoc*; if user `Bob` forwards the message of Figure 2.15 with an added note to someone in yet another naming context, any unqualified addresses in the added note would need a different explicit context reference. Although this scheme is not actually used in any e-mail system that the authors are aware of, it has been used in other naming systems. An example is the base element of HTML, the display language of the World Wide Web, described briefly in Section 3.2.2.

A closely related problem is that different contexts may bind different names for the same object. For example, to call a certain telephone, it may be that a person in the same organization dials 2-7104, a second person across the city dials 312-7104, a third who is a little farther away dials (517) 312-7104, and a person in another country may have to dial 001 (517) 312-7104. When the same object has different names in different contexts, passing a name from one user to another is awkward because, as with the e-mail message example, someone must translate the name before the other user can use it. As with the e-mail address, if someone hands you a scrap of paper on

```
To: Bob
cc: Charlies@cse.Scholarly.edu
From: Alice
Context: Pedantic.edu
------
Based on Dawn's suggestions, this chapter has
experienced a major overhaul this year. If you
like it, send your compliments to Dawn (her e-mail
address is "Dawn"); if you do not like it, send your
complaints to me.
```

FIGURE 2.15
An e-mail message that provides an explicit context reference as one of its headers.

which is written the telephone number 312–7104, simply dialing that number may or may not ring the intended telephone. Even though the several names are related, some effort may be required to figure out just what translation is required.

2.2.3 Path Names, Naming Networks, and Recursive Name Resolution

The second of the three common name-mapping algorithms listed on page 64 is *recursive name resolution*. A path name can be thought of as a name that explicitly includes a reference to the context in which it should be resolved. In some naming schemes, path names are written with the context reference first, in others with the context reference last. Some examples of path names are:

```
ginger.pedantic.edu.
/usr/bin/emacs
Macintosh HD:projects:CSE 496:problem set 1
Chapter 2, section 2, part 3, first paragraph
Paragraph 1 of part 3 of section 2 of chapter 2
```

As these examples suggest, a path name involves multiple components and some syntax that permits a name resolver to parse the components. The last two examples illustrate that different naming schemes place the component names in opposite orders, and indeed the other examples also demonstrate both orders. The order of the components must be known to the user of the name and to the name resolver, but either way interpretation of the path name is most easily explained recursively by borrowing terminology from the representation of numbers: all but the least significant component of a path name is an explicit context reference that identifies the context to be used to resolve that least significant component. In the above examples, the least significant components and their explicit context references are, respectively,

Least significant component	Explicit context reference
`ginger`	`pedantic.edu.`
`emacs`	`/usr/bin`
`problem set 1`	`Macintosh hd:projects:CSE 491`
first paragraph	Chapter 2, section 2, part 3
Paragraph 1	part 3 of section 2 of chapter 2

The recursive aspect of this description is that the explicit context reference is itself a path name that must be resolved. So we repeat the analysis as many times as needed until what was originally the most significant component of the path name is also the least significant component, at which point the resolver can do an ordinary table lookup using some context. In the choice of this context, the previous discussion of default and explicit context references again applies. In a typical design, the resolver uses one of two default context references:

- A special context reference, known as the *root,* that is built in to the resolver. The root is an example of a universal name space. A path name that the resolver can resolve with recursion that ends at the root context is known as an *absolute path name*.
- The path name of yet another default context. To avoid circularity, this path name must be an absolute path name. A path name that is resolved by looking up its most significant component in yet another context is known as a *relative path name*. (In a file system, the path name of this default context is what example 4 on page 68 identified as the working directory.) Thus in the UNIX file system, for example, if the working directory is `/usr/Alice`, the relative path name `plans/Monday` would resolve to the same file as the absolute path name `/usr/Alice/plans/Monday`.

If a single name resolver is prepared to resolve both relative and absolute path names, some scheme such as a syntactic flag (e.g., the initial "`/`" in `/usr/bin/emacs` and the terminal "`.`" in `ginger.pedantic.edu`.) may distinguish one from the other, or perhaps the name resolver will try both ways in some order, using the first one that seems to work. Trying two schemes in order is a simple form of multiple name lookup, about which we will have more to say in the next subsection.

Path names can also be thought of as identifying objects that are organized in what is called a *naming network*. In a naming network, contexts are treated as objects, and any context may contain a name-to-object binding for any other object, including another context. The name resolver somehow chooses one context to use as the root (perhaps by having a lower-level name for that context wired into the resolver), and it then resolves all absolute path names by tracing a path from the chosen root to the first named context in the path name, then the next, continuing until it reaches the object that was named by the original path name. It similarly resolves relative path names starting with a default context found in a variable in its environment. That variable contains the absolute path name of the default context. Since there can be many paths from one place to another, there can be many different path names for the same object or context. Multiple names for the same object are known as *synonyms* or *aliases*. (This text avoids the word "alias" because different systems use it in quite different ways.) On the other hand, since the root provides a universal name space, every object that uses the same absolute path name is referring to the same exporting object.

Sharing names of a naming network can be a problem because each user may express path names relative to a different starting point. As a result, it may not be

obvious how to translate a path name when passing it from one user to another. One standard solution to this problem is to require that users share only absolute path names, all of which begin with the root.

The file system of a computer operating system is usually organized as a naming network, with directories acting as contexts. It is common in file systems to encounter implementation-driven restrictions on the shape of the naming network, for example, requiring that the contexts be organized in a *naming hierarchy* with the root acting as the base of the tree. A true naming hierarchy is so constraining that it is rarely found in practice; real systems, even if superficially hierarchical, usually provide some way of adding cross-hierarchy *links.* The simplest kind of link is just a synonym: a single object may be bound in more than one context. Some systems allow a more sophisticated kind of link, known as an *indirect name*. An indirect name is one that a context binds to another name in the same name space rather than to an object. Because many designers have independently realized that indirect names are useful, they have come to be called by many different labels, including *symbolic link*, *soft link, alias,* and *shortcut*. The UNIX file system described in Section 2.5 includes a naming hierarchy, links, and indirect names called soft links.

A path name has internal structure, so a naming scheme that supports path names usually has rules regarding construction of allowable path names. Path names may have a maximum length, and certain symbols may be restricted for use only as structural separators.

2.2.4 Multiple Lookup: Searching through Layered Contexts

Returning to the topic of default contexts (in the taxonomy of Figure 2.12), context assignment rules are a blunt tool. For example, a directory containing library programs may need to be shared among different users; no single assignment rule can suffice. This inflexibility leads to the third, more elaborate name resolution scheme, *multiple lookup.*[*] The idea of multiple lookup is to abandon the notion of a single, default context and instead resolve the name by systematically trying several different contexts. Since a name may be bound in more than one context, multiple lookup can produce multiple resolutions, so some scheme is needed to decide which resolution to use.

A common such scheme is called the *search path*, which is nothing more than a specific list of contexts to be tried, in order. The name resolver tries to resolve the name using the first context in the list. If it gets a not-found result, it tries the next context, and so on. If the name is bound in more than one of the listed contexts, the one earliest in the list wins and the resolver returns the value associated with that binding.

A search path is often used in programming systems that have libraries. Suppose, for example, a library procedure that calculates the square root math function exports

[*]The operating system community traditionally uses the word "search" for multiple lookup, but the advent of "search engines" on both the Internet and the desktop has rendered that usage ambiguous. The last paragraph of Section 2.2.4, on page 75, discusses this topic.

a procedure interface named SQRT. After compiling this function, the writer places a copy of the binary program in a math library. A prospective user of the square root function writes the statement

$x \leftarrow \text{SQRT}(y)$

in a program, and the compiler generates code that uses the procedure named SQRT. The next step is that the compiler (or in some systems a later loader) undertakes a series of lookups in various public and private libraries that it knows about. Each library is a context, and the search path is a list of the library contexts. Each step of the multiple lookup involves an invocation of a simpler, single-context name resolver. Some of these attempted resolutions will probably return a not-found result. The first resolution attempt that finds a program named SQRT will return that program as the result of the lookup.

A search path is usually implemented as a per-user list, some or all of whose elements the user can set. By placing a library that contains personally supplied programs early in the search path, an individual user can effectively replace a library program with another that has the same name, thereby providing a *user-dependent binding*. This replace-by-name feature can be useful, but it can also be hazardous because one may unintentionally choose a name for a program that is also exported by some completely unrelated library program. When some other application tries to call that unrelated program, the ensuing multiple lookup may find the wrong one. As the number of libraries and names in the search path increases, the chance increases that two libraries will accidentally contain two unrelated programs that happen to export the same name.

Despite the hazards, search paths are a widely used mechanism. In addition to loaders using search paths to locate library procedures, user interfaces use search paths to locate commands whose names the user typed, compilers use search paths to locate interfaces, documentation systems use search paths to find cited documents, and word processing systems use search paths to locate text fragments to be included in the current document.

Some naming schemes use a more restricted multiple lookup method. For example, rather than allowing an arbitrary list of contexts, a naming scheme may require that contexts be arranged in nested layers. Whenever a resolution returns not-found in some layer, the resolver retries in the enclosing layer. Layered contexts were at one time popular in programming languages, where programs define and call on subprograms, because it can be convenient (to the point of being undisciplined, which is why it is no longer so popular) to allow a subprogram access by name to the variables of the defining or calling program. For another example, the scheme for numbering Internet network attachment points has an outer public layer and an inner private layer. Certain Internet address ranges (e.g., all addresses with a first byte of 10) are reserved for use in private networks; those address ranges constitute an inner private layer. These network addresses may be bound to different network attachment points in different private contexts without risk of conflict. Internet addresses that are outside the ranges reserved for private contexts should not be bound in any private context; they are instead resolved in the public context.

In a set of layered contexts, the *scope* of a name is the range of layers in which the name is bound to the same object. A name that is bound only in the outermost layer, and is always bound to the same object, independent of the current context layer, is known as a *global name*. The outermost layer that resolves global names is an example of a universal name space.

Incidentally, we have now used the term *path* as both an adjective qualifier and a noun, but with quite different meanings. A *path name* is a name that carries its own explicit context, while a *search path* is a context that consists of a list of contexts. Thus each element of a search path may be a path name.

The word "search" also has another, related but somewhat different, meaning. Internet search engines such as Google and AltaVista take as input a query consisting of one or more key words, and they return a list of World Wide Web pages that contain those key words. Multiple results (known as "hits") are the common case, and Google, for example, implements a sophisticated system for ranking the hits. Google also offers the user the choice of receiving just the highest-ranked hit ("I'm feeling lucky") or receiving a rank-ordered list of hits. Most modern desktop computer systems also provide some form of key word search for local files. When one encounters the unqualified word "search", it is a good idea to pause and figure out whether it refers to multiple lookup or to key word query.

2.2.5 Comparing Names

As mentioned earlier, one more operation is sometimes applied to names:

result ← COMPARE(*name1*, *name2*)

where *result* is a binary value, TRUE or FALSE. The meaning of name comparison requires some thought because the invoker might have one of three different questions in mind:

1. Are the two names the same?
2. Are the two names bound to the same value?
3. If the value or values are actually the identifiers of storage containers, such as memory cells or disk sectors, are the contents of the storage containers the same?

The first question is mechanically easiest to answer because it simply involves comparing the representations of the two names ("Is Jim Smith the same as Jim Smith?"), and it is exactly what a name resolver does when it looks things up in a table-lookup context: look through the context for a name that is the same as the one being resolved. On the other hand, in many situations the answer is not useful, since the same name may be bound to different values in different contexts and two different names may be synonyms that are bound to the same value. All one learns from the first question is whether or not the name strings have the same bit pattern.

For that reason, the answer to the second question is often more interesting. ("Is the Jim Smith who just received the Nobel prize the same Jim Smith I knew in high school?") Getting that answer requires supplying the contexts for the two names as

additional arguments to COMPARE, so that it can resolve the names and compare the results. Thus, for example, resolving the variable name *A* and the variable name *B* may reveal that they are both bound to the same storage cell address. Even this answer may still not reveal as much as expected because the two names may resolve to two names of a different, lower-layer naming scheme, in which case the same questions need to be asked recursively about the lower-layer names. For example, variable names *A* and *B* may be bound to different storage cell addresses, but if a virtual memory is in use those different virtual storage cell addresses might map to the same physical cell address. (This example will make more sense when we reach Chapter 5.)

Even after reaching the bottom of that recursion, the result may be the names of two different physical storage containers that contain identical copies of data, or it may be two different lower-layer names (that is, synonyms) for the same storage container. ("This biography file on Jim Smith is identical to that biography file on Jim Smith. Are there one or two biography files?" "This biography about Edwin Aldrin is identical to that biography about Buzz Aldrin. Are those two names for the same person?") Thus the third question arises, along with a need to understand what it means to be the "same". Unless one has some specific understanding of the underlying physical representation, the only way to distinguish the two cases may be to change the contents of one of the named storage containers and see if that causes the contents of the other one to change. ("Kick this one and see if that one squeals.")

In practice, systems (and some programming languages) typically provide several COMPARE operators that have different semantics designed to help answer these different questions, and the programmer or user must understand which COMPARE operation is appropriate for the task at hand. For example, the LISP language provides three comparison operators, named EQ (which compares the bindings of its named arguments), EQU (which compares the values of its named arguments), and EQUALS (which recursively compares entire data structures.)

2.2.6 Name Discovery

Underlying all name reference is a recursive protocol that answers the question, "How did you know to use this name?" This *name discovery protocol* informs an object's prospective user of the name that the object exports. Name discovery involves two basic elements: the exporter *advertises* the existence of the name, while the prospective user *searches* for an appropriate advertisement. The thing that makes name discovery recursive is that the name user must first know the name of a place to search for the advertisement. This recursion must terminate somewhere, perhaps in a direct, outside-the-computer communication between some name user and some name exporter.

The simplest case is a programmer who writes a program consisting of two procedures, one of which refers to the other by name. Since the same programmer wrote both, name discovery is explicit and no recursion is necessary. Next, suppose the two programs are written by two different programmers. The programmer who wants to use a procedure by name must somehow discover the exported name. One possibility is that the second programmer performs the advertisement by shouting the

procedure's name down the hall. Another possibility is that the using programmer looks in a shared directory in which everyone agrees to place shared procedures. How does that programmer know the name of that shared directory? Perhaps someone shouted that name down the hall. Or perhaps it is a standard library directory whose name is listed in the programmers' reference manual, in which case that manual terminates the recursive protocol. Although program library names don't usually appear in magazine advertisements or on billboards, it has become commonplace to discover the name of a World Wide Web site in such places. Name discovery can take several forms:

- *Well-known name:* A name (such as "Google" or "Yahoo!") that has been advertised so widely that one can depend on it being stable for at least as long as the thing it names. Running across a well-known name is a method of name discovery.
- *Broadcast:* A way of advertising a name, for example by wearing a badge that says "Hello, my name is . . . ", posting the name on a bulletin board, or sending it to a mailing list. Broadcast is used by automatic configuration protocols sometimes called "plug-and-play" or "zero configuration". It may even be used on a point-to-point communication link in the hope that there is someone listening at the other end who will reply. Listening for broadcasts is a method of name discovery.
- *Query* (also called *search*): Present one or more key words to, for example, a search engine such as Google. Query is a widely used method of name discovery.
- *Broadcast query:* A generalized form of key word query. Ask everyone within hearing distance "does anyone know a name for . . . ?" (sometimes confusingly called "reverse broadcast").
- *Resolving a name of one name space to a name of a different name space:* Looking up a name in the telephone book leads to discovery of a telephone number. The Internet Domain Name System, described in Section 4.4, performs a similar service, looking up a domain name and returning a network attachment point address.
- *Introduction:* What happens at parties and in on-line dating services. Some entity that you already know knows a name and gives that name to you. In a computer system, a friend may send you an e-mail message that mentions the name of an interesting Web site or the e-mail address of another friend. For another example, each World Wide Web page typically contains introductions (technically known as hypertext links) to other Web pages.
- *Physical rendezvous:* A meeting held outside the computer. It requires somehow making prior arrangements concerning time and place, which implies prior communication, which implies prior knowledge of some names. Once set up, physical rendezvous can be used for discovering other names as well as for verifying authenticity. Many organizations require that setting up a new account on a company computer system must involve a physical rendezvous with the system administrator to exchange names and choose a password.

Any of the above methods of name discovery may require first discovering some other name, such as the name of the reference source for well-known names, the name of the bulletin board on which broadcasts are placed, the name of the name resolver, the name of the party host, and so on. The method of discovering this other name may be the same as the method first invoked, or it may be different. The important thing the designer must keep in mind is that the recursion must terminate somewhere—it can't be circular.

Some method of name discovery is required wherever a name is needed. An interesting exercise is to analyze some of the examples of names mentioned in earlier parts of this chapter, tracing the name discovery recursion to see how it terminates, because in many cases that termination is so distant from the event of name usage and resolution that it has long since been forgotten. Many additional examples of name discovery will show up in later chapters: names used for clients and services, where a client needs to discover the name of an appropriate service; data communication networks, where routing provides a particularly explicit example of name discovery; and security, where it is critical to establish the integrity of the terminating step.

2.3 ORGANIZING COMPUTER SYSTEMS WITH NAMES AND LAYERS

Section 2.1 demonstrated how computer system designers use layers to implement more elaborate versions of the three fundamental abstractions, and Section 2.2 explained how names are used to connect system components. Designers also use layers and names in many other ways in computer systems. Figure 2.16 shows the typical organization of a computer system as three distinct layers. The bottom layer consists of hardware components, such as processors, memories, and communication links. The middle layer consists of a collection of software modules, called the *operating system* (see Sidebar 2.4), that abstract these hardware resources into a convenient *application programming interface (API)*. The top layer consists of software that implements application-specific functions, such as a word processor, payroll program, computer game, or Web

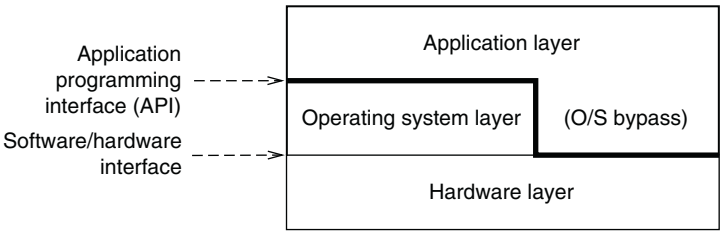


FIGURE 2.16
A typical computer system organized in three layers. The operating system layer allows bypass, so the application layer can directly invoke many features of the hardware layer. However, the operating system layer hides certain dangerous features of the hardware layer.

Sidebar 2.4 What is an Operating System? An *operating system* is a set of programs and libraries that make it easy for computer users and programmers to do their job. In the early days of computers, operating systems were simple programs that assisted operators of computers (at that time the only users who interacted with a computer directly), which is why they are called operating systems.

Today operating systems come in many flavors and differ in the functions they provide. The operating system for the simplest computers, such as that for a microwave oven, may comprise just a library that hides hardware details in order to make it easier for application programmers to develop applications. Personal computers, on the other hand, ship with operating systems that contain tens of millions of lines of code. These operating systems allow several people to use the same computer; permit users to control which information is shared and with whom; can run many programs at the same time while keeping them from interfering with one another; provide sophisticated user interfaces, Internet access, file systems, backup and archive applications, device drivers for the many possible hardware gadgets on a personal computer, and a wide range of abstractions to simplify the job of application programmers, and so on.

Operating systems also offer an interesting case study of system design. They are evolving rapidly because of new requirements. Their designers face a continuous struggle to control their complexity. Some modern operating systems have interfaces consisting of thousands of procedures, and their implementations are so complex that it is a challenge to make them work reliably.

This book has much more to say about operating systems, starting in Section 5.1.1, where it begins development of a minimal model operating system.

browser. If we examine each layer in detail, we are likely to find that it is itself organized in layers. For example, the hardware layer may comprise a lower layer of gates, flip-flops, and wires, and an upper layer of registers, memory cells, and finite-state machines.

The exact division of labor between the hardware layer and the software layers is an engineering trade-off and a topic of considerable debate between hardware and software designers. In principle, every software module can be implemented in hardware. Similarly, most hardware modules can also be implemented in software, except for a few foundational components such as transistors and wires. It is surprisingly difficult to state a generic principle for how to decide between an implementation in hardware or software. Cost, performance, flexibility, convenience, and usage patterns are among the factors that are part of the trade-off, but for each individual function they may be weighted differently. Rather than trying to invent a principle, we discuss the trade-off between hardware and software in the context of specific functions as they come up.

The operating system layer usually exhibits an interesting phenomenon that we might call *layer bypass*. Rather than completely hiding the lower, hardware layer, an operating system usually hides only a few features of the hardware layer, such as

particularly dangerous instructions. The remaining features of the hardware layer (in particular, most of the instruction repertoire of the underlying processor) pass through the operating system layer for use directly by the application layer, as in Figure 2.16. Thus, the dangerous instructions can be used only by the operating system layer, while all of the remaining instructions can be used by both the operating system and application layers. Conceptually, a designer could set things up so that the operating system layer intercepts every invocation of the hardware layer by the application layer and then explicitly invokes the hardware layer. That design would slow a heavily used interface down unacceptably, so in the usual implementation the application layer directly invokes the hardware layer, completely bypassing the operating system layer. Operating systems provide bypass for performance reasons, but bypass is not unique to operating systems, nor is it used only to gain performance. For example, the Internet is a layered communication system that permits bypass of most features of most of its layers, to achieve flexibility.

In this section we examine two examples of layered computer system organization: the hardware layer at the bottom of a typical computer system and one part of the operating system layer that creates the typical application programming interface known as the file system.

2.3.1 A Hardware Layer: The Bus

The hardware layer of a typical computer is constructed of modules that directly implement low-level versions of the three fundamental abstractions. In the example of Figure 2.17, the processor modules interpret programs, the random access memory modules store both programs and data, and the input/output (I/O) modules implement communication links to the world outside the computer.

There may be several examples of each kind of hardware module—multiple processors (perhaps several on one chip, an organization that goes by the buzzword name *multicore*), multiple memories, and several kinds of I/O modules. On closer inspection the I/O modules turn out to be specialized interpreters that implement I/O programs. Thus, the disk controller is an interpreter of disk I/O programs. Among its duties are mapping disk addresses to track and sector numbers and moving data from the disk to the memory. The network controller is an interpreter that talks on its other side to one or more real communication links. The display controller interprets display lists that it finds in memory, lighting pixels on the display as it goes. The keyboard controller interprets keystrokes and places the result in memory. The clock may be nothing but a minuscule interpreter that continually updates a single register with the time of day.

The various modules plug into the shared *bus*, which is a highly specialized communication link used to `SEND` messages to other modules. There are numerous bus designs, but they have some common features. One such common feature is a set of wires*

*This description in terms of several parallel wires is of a structure called a *parallel bus*. A more thorough discussion of link communication protocols in Section 7.3 [on-line] shows how a bus can also be implemented by sending coded signals down just a few wires, a scheme called a *serial bus*.

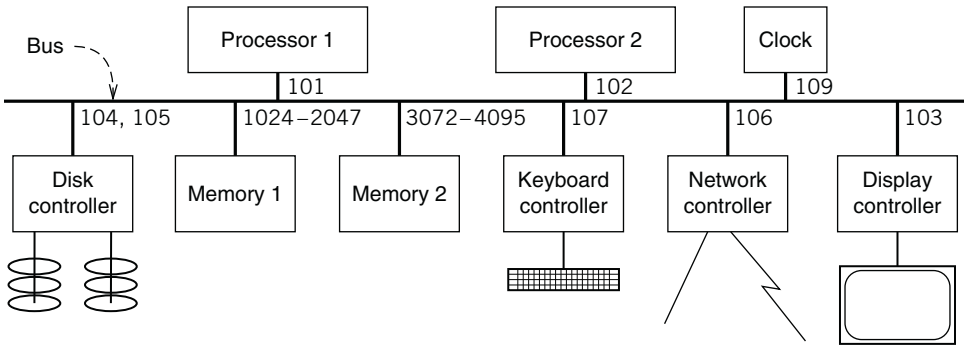


FIGURE 2.17

A computer with several modules connected by a shared bus. The numbers are the bus addresses to which the attached module responds.

comprising address, data, and control lines that connect to a *bus interface* on each module. Because the bus is shared, a second common feature is a set of rules, called the *bus arbitration protocol,* for deciding which module may send or receive a message at any particular time. Some buses have an additional module, the *bus arbiter*, a circuit or a tiny interpreter that chooses which of several competing modules can use the bus. In other designs, bus arbitration is a function distributed among the bus interfaces. Just as there are many bus designs, there are also many bus arbitration protocols. A particularly influential example of a bus is the UNIBUS®, introduced in the 1970s by Digital Equipment Corporation. The modularity provided by a shared bus with a standard arbitration protocol helped to reshape the computer industry, as was described in Sidebar 1.5.

A third common feature of bus designs is that a bus is a *broadcast* link, which means that every module attached to the bus hears every message. Since most messages are actually intended for just one module, a field of the message called the *bus address* identifies the intended recipient. The bus interface of each module is configured to respond to a particular set of bus addresses. Each module examines the bus address field (which in a parallel bus is usually carried on a set of wires separate from the rest of the message) of every message and ignores any message not intended for it. The bus addresses thus define an address space. Figure 2.17 shows that the two processors might accept messages at bus addresses 101 and 102, respectively; the display controller at bus address 103; the disk controller at bus addresses 104 and 105 (using two addresses makes it convenient to distinguish requests for its two disks); the network at bus address 106; the keyboard at bus address 107; and the clock at bus address 109. For speed, memory modules typically are configured with a range of bus addresses, one bus address per memory address. Thus, if in Figure 2.17 the two memory modules each implement an address space of 1,024 memory addresses, they might be configured with bus addresses 1024–2047 and 3072–4095, respectively.*

*These bus addresses are chosen for convenience of the illustration. In practice, a memory module is more likely to be configured with enough bus addresses to accommodate several gigabytes.

Any bus module that wishes to send a message over the bus must know a bus address that the intended recipient is configured to accept. Name discovery in some buses is quite simple: whoever sets up the system explicitly configures the knowledge of bus addresses into the processor software, and that software passes this knowledge along to other modules in messages it sends over the bus. Other bus designs dynamically assign bus addresses to modules as they are plugged in to the bus and announce their presence.

A common bus design is known as *split-transaction*. In this design, when one module wants to communicate with another, the first module uses the bus arbitration protocol on the control wires to request exclusive use of the bus for a message. Once it has that exclusive use, the module places a bus address of the destination module on the address wires and the remainder of the message on the data wires. Assuming a design in which the bus and the modules attached to it run on uncoordinated clocks (that is, they are asynchronous), it then signals on one of the control wires (called READY) to alert the other modules that there is a message on the bus. When the receiving module notices that one of its addresses is on the address lines of the bus, it copies that address and the rest of the message on the data wires into its local registers and signals on another control line (called ACKNOWLEDGE) to tell the sender that it is safe to release the bus so that other modules can use it. (If the bus and the modules are all running with a common clock, the READY and ACKNOWLEDGE lines are not needed; instead, each module checks the address lines on each clock cycle.) Then, the receiver inspects the address and message and performs the requested operation, which may involve sending one or more messages back to the original requesting module or, in some cases, even to other modules.

For example, suppose that processor #2, while interpreting a running application program, encounters the instruction

```
LOAD        1742, R1
```

which means "load the contents of memory address 1742 into processor register R1". In the simplest scheme, the processor just translates addresses it finds in instructions directly to bus addresses without change. It thus sends this message across the bus:

processor #2 ⇒ *all bus modules*: {1742, READ, 102}

The message contains three fields. The first message field (1742) is one of the bus addresses to which memory #1 responds; the second message field requests the recipient to perform a READ operation; and the third indicates that the recipient should send the resulting value back across the bus, using the bus address 102. The memory addresses recognized by each memory module are based on powers of two, so the memory modules can recognize all of the addresses in their own range by examining just a few high-order address bits. In this case, the bus address is within the range recognized by memory module 1, so that module responds by copying the message into its own registers. It acknowledges the request, the processor releases the bus, and the memory module then performs the internal operation

value ← READ (1742)

With *value* in hand, the memory module now itself acquires the bus and sends the result back to processor #2 by performing the bus operation

memory #1 ⇒ *all bus modules*: {102, *value*}

where 102 is the bus address of the processor as supplied in the original READ request message. The processor, which is probably waiting for this result, notices that the bus address lines now contain its own bus address 102. It therefore copies the value from the data lines into its register R1, as the original program instruction requested. It acknowledges receipt of the message, and the memory module releases the bus for use by other modules.

Simple I/O devices, such as keyboards, operate in a similar fashion. At system initialization time, one of the processors SENDs a message to the keyboard controller telling it to SEND all keystrokes to that processor. Each time that the user depresses a key, the keyboard controller SENDs a message to the processor containing as data the name of the key that was depressed. In this case, the processor is probably *not* waiting for this message, but its bus interface (which is in effect a separate interpreter running concurrently with the processor) notices that a message with its bus address has appeared. The bus interface copies the data from the bus into a temporary register, acknowledges the message, and sends a signal to the processor that will cause the processor to perform an interrupt on its next instruction cycle. The interrupt handler then transfers the data from the temporary register to some place that holds keyboard input, perhaps by SENDing yet another message over the bus to one of the memory modules.

One potential problem of this design is that the interrupt handler must respond and read the keystroke data from the temporary register before the keyboard handler SENDs another keystroke message. Since keyboard typing is slow compared with computer speeds, there is a good chance that the interrupt handler will be there in time to read the data before the next keystroke overwrites it. However, faster devices such as a hard disk might overwrite the temporary register. One solution would be to write a processor program that runs in a tight loop, waiting for data that the disk controller sends over the bus and immediately SENDing that data again over the bus to a memory module.

Some low-end computer designs do exactly that, but a designer can obtain substantially higher performance by upgrading the disk controller to use a technique called *direct memory access*, or DMA. With this technique, when a processor SENDs a request to a disk controller to READ a block of data from the disk, it includes the address of a buffer in memory as a field of the request message. Then, as data streams in from the disk, the disk controller SENDs it directly to the memory module, incrementing the memory address appropriately between SENDs. In addition to relieving the load on the processor, DMA also reduces the load on the shared bus because it transfers each piece of data across the bus just once (from the disk controller to the memory) rather than twice (first from the disk controller to the processor and then from the processor to the memory). Also, if the bus allows long messages, the DMA controller may be able to take better advantage of that feature than the processor, which is usually designed to SEND and RECEIVE bus data in units that are the same size as its own registers. By SENDing longer messages, the DMA controller increases performance

because it amortizes the overhead of the bus arbitration protocol, which it must perform once per message. Finally, DMA allows the processor to execute some other program at the same time that the disk controller is transferring data. Because concurrent operation can hide the latency of the disk transfer, it can provide an additional performance enhancement. The idea of enhancing performance by hiding latency is discussed further in Chapter 6.

A convenient interface to I/O and other bus-attached modules is to assign bus addresses to the control registers and buffers of the module. Since each processor maps bus addresses directly into its own memory address space, LOAD and STORE instructions executed in the processor can in effect address the registers and buffers of the I/O module as if they were locations in memory. The technique is known as *memory-mapped I/O.*

Memory-mapped I/O can be combined with DMA. For example, suppose that a disk controller designed for memory-mapped I/O assigns bus addresses to four of its control registers as follows:

bus address	control register
121	*sector_number*
122	*DMA_start_address*
123	*DMA_count*
124	*control*

To do disk I/O, the processor uses STORE instructions to SEND appropriate initialization values to the first three disk controller registers and a final STORE instruction to SEND a value that sets a bit in the *control* register that the disk controller interprets as the signal to start. A program to GET a 256-byte disk sector currently stored at sector number 11742 and transfer the data into memory starting at location 3328 starts by loading four registers with these values and then issuing STOREs of the registers to the appropriate bus addresses:

```
R1 ← 11742; R2 ← 3328; R3 ← 256; R4 ← 1;
STORE 121,R1                    // set sector number
STORE 122,R2                    // set memory address register
STORE 123,R3                    // set byte count
STORE 124,R4                    // start disk controller running
```

Upon completion of the bus SEND generated by the last STORE instruction, the disk controller, which was previously idle, leaps into action, reads the requested sector from the disk into an internal buffer, and begins using DMA to transfer the contents of the buffer to memory one block at a time. If the bus can handle blocks that are 8 bytes long, the disk controller would SEND a series of bus messages such as

```
disk controller #1 ⇒ all bus modules:        {3328, block[1]}
disk controller #1 ⇒ all bus modules:        {3336, block[2]}
etc . . .
```

Memory-mapped I/O is a popular interface because it provides a uniform memory-like LOAD and STORE interface to every bus module that implements it. On the other hand,

the designer must be cautious in trying to extend the memory-mapped model too far. For example, trying to arrange so the processor can directly address individual bytes or words on a magnetic disk could be problematic in a system with a 32-bit address space because a disk as small as 4 gigabytes would use up the entire address space. More important, the latency of a disk is extremely large compared with the cycle time of a processor. For the STORE instruction to sometimes operate in a few nanoseconds (when the address is in electronic memory) and other times require 10 milliseconds to complete (when the address is on the disk) would be quite unexpected and would make it difficult to write programs that have predictable performance. In addition, it would violate a fundamental rule of human engineering, the ***principle of least astonishment*** (see Sidebar 2.5). The bottom line is that the physical properties of the magnetic disk make the DMA access model more appropriate than the memory-mapped I/O model.

Sidebar 2.5 Human Engineering and the Principle of Least Astonishment An important principle of human engineering for usability, which for computer systems means designing to make them easy to set up, easy to use, easy to program, and easy to maintain, is the principle of least astonishment.

The principle of least astonishment

People are part of the system. The design should match the user's experience, expectations, and mental models.

Human beings make mental models of the behavior of everything they encounter: components, interfaces, and systems. If the actual component, interface, or system follows that mental model, there is a better chance that it will be used as intended and less chance that misuse or misunderstanding will lead to a mistake or disappointment. Since complexity is relative to understanding, the principle also tends to help reduce complexity.

For this reason, when choosing among design alternatives, it is usually better to choose one that is most likely to match the expectations of those who will have to use, apply, or maintain the system. The principle should also be a factor when evaluating tradeoffs. It applies to all aspects of system design, especially to the design of human interfaces and to computer security.

Some corollaries are to be noted: Be consistent. Be predictable. Minimize side-effects. Use names that describe. Do the obvious thing. Provide sensible interpretations for all reasonable inputs. Avoid unnecessary variations.

Some authors prefer the words "principle of least surprise" to "principle of least astonishment". When Bayesian statisticians invoke the principle of least surprise, they usually mean "choose the mostly likely explanation", a version of the closely related Occam's razor. (See the aphorism at the bottom of page 9.)

(*Sidebar continues*)

Human Engineering and the Original Murphy's Law. If you ask a group of people "What is Murphy's law?" most responses will be some variation of "If anything can go wrong, it will", followed by innumerable equivalents, such as the toast always falls butter side down.

In fact, Murphy originally said something quite different. Rather than a comment on the innate perversity of inanimate objects (sometimes known as *Finagle's law*, from a science fiction story), Murphy was commenting on a property of human nature that one must take into account when designing complex systems: *If you design it so that it can be assembled wrong, someone will assemble it wrong*. Murphy was pointing out the wisdom of good human engineering of things that are to be assembled: design them so that the only way to assemble them is the right way.

Edward A. Murphy, Jr., was an engineer working on United States Air Force rocket sled experiments at Edwards Air Force Base in 1949, in which Major John Paul Stapp volunteered to be subjected to extreme decelerations (40 Gs) to determine the limits of human tolerance for ejection seat design. On one of the experiments, someone wired up all of the strain gauges incorrectly, so at the end of Stapp's (painful) ride there was no usable data. Murphy said, in exasperation at the technician who wired up the strain gauges, "if that guy can find a way to do it wrong, he will." Stapp, who as a hobby made up laws at every opportunity, christened this observation "Murphy's law," and almost immediately began telling it to others in the different and now widely known form "If anything can go wrong, it will."

A good example of Murphy's original observation in action showed up in an incident on a Convair 580 cargo plane in 1997. Two identical control cables ran from a cockpit control to the elevator trim tab, a small movable surface on the rear stabilizing wing that, when adjusted up or down, forces the nose of the plane to rise or drop, respectively. Upon take-off on the first flight after maintenance, the pilots found that the plane was pitching nose-up. They tried adjusting the trim tab to maximum nose-down position, but the problem just got worse. With much effort they managed to land the plane safely. When mechanics examined the plane, they discovered that the two cables to the trim tab had been interchanged, so that moving the control up caused the trim tab to go down and vice versa[*].

A similar series of incidents in 1988 and 1989 involved crossed connections in cargo area smoke alarm signal wires and fire extinguisher control wires in the Boeing 737, 757, and 767 aircraft[†].

---

[*]Transportation Safety Board of Canada, *Report A97O0077*, January 13, 2000, updated October 6, 2002.

[†]Karen Fitzgerald, "Boeing's crossed connections", *IEEE Spectrum 26*, 5 (May 1989), pages 30–35.

2.3.2 A Software Layer: The File Abstraction

The middle and higher layers of a computer system are usually implemented as software modules. To make this layered organization concrete, consider the *file*, a high-level version of the memory abstraction. A file holds an array of bits or bytes, the number of which the application chooses. A file has two key properties:

- *It is durable.* Information, once stored, will remain intact through system shutdowns and can be retrieved later, perhaps weeks or months later. Applications use files to durably store documents, payroll data, e-mail messages, programs, and anything else they do not want to be lost.
- *It has a name.* The name of a file allows users and programs to store information in such a way that they can find and use it again at a later time. File names also make it possible for users to share information. One can WRITE a named file and tell a friend the file name, and then the friend can use the name to READ the file.

Taken together, these two features mean that if, for example, Alice creates a new file named "strategic plan", WRITEs some information in it, shuts down the computer, and the next day turns it on again, she will then be able to READ the file named "strategic plan" and get back its content. Furthermore, she can tell Bob to look at the file named "strategic plan". When Bob asks the system to READ a file with that name, he will read the file that she created. Most file systems also provide other additional properties for files, such as timestamps to determine when they were created, last modified, or last used, assurances about their durability (a topic that Chapter 10 [on-line] revisits), and the ability to control who may share them (one of the topics of Chapter 11 [on-line]).

The system layer implements files using modules from the hardware layer. Figure 2.18 shows the pseudocode of a simple application that reads input from a keyboard device, writes that input to a file, and also displays it on the display device.

```
character buf                                          // buffer for input character
file ← OPEN ("strategic plan", READWRITE)              // open file for reading and writing
input ← OPEN ("keyboard", READONLY)                    // open keyboard device for reading
display ← OPEN ("display", WRITEONLY)                  // open display device for writing

while not END_OF_FILE (input) do
    READ (input, buf, 1)                               // read 1 character from keyboard
    WRITE (file, buf, 1)                               // store input into file
    WRITE (display, buf, 1)                            // display input

CLOSE (file)
CLOSE (input)
CLOSE (display)
```

FIGURE 2.18

Using the file abstraction to implement a display program, which also writes the keyboard input in a file. For clarity, this program ignores the possibility that any of the abstract file primitives may return an error status.

A typical API for the file abstraction contains calls to OPEN a file, to READ and WRITE parts of the file, and to CLOSE the file. The OPEN call translates the file name into a temporary name in a local name space to be used by the READ and WRITE operations. Also, OPEN usually checks whether this user is permitted access to the file. As its last step, OPEN sets a *cursor*, sometimes called a *file pointer*, to zero. The cursor records an offset from the beginning of the file to be used as the starting point for READs and WRITEs. Some file system designs provide a separate cursor for READs and WRITEs, in which case OPEN may initialize the WRITE cursor to the number of bytes in the file.

A call to READ delivers to the caller a specified number of bytes from the file, starting from the READ cursor. It also adds to the READ cursor the number of bytes read so that the next READ proceeds where the previous READ left off. If the program asks to read bytes beyond the end of the file, READ returns some kind of end-of-file status indicator.

Similarly, the WRITE operation takes as arguments a buffer with bytes and a length, stores those bytes in the file starting at the offset indicated by the WRITE cursor (if the WRITE cursor starts at or reaches the end of the file, WRITE usually implies extending the size of the file), and adds to the WRITE cursor the number of bytes written so that the next WRITE can continue from there. If there is not enough space on the device to write that many bytes, the WRITE procedure fails by returning some kind of device-full error status or exception.

Finally, when the program is finished reading and writing, it calls the CLOSE procedure. CLOSE frees up any internal state that the file system maintains for the file (for example, the cursors and the record of the temporary file name, which is no longer meaningful). Some file systems also ensure that, when CLOSE returns, all parts of the modified file have been stored durably on a non-volatile memory device. Other file systems perform this operation in the background after CLOSE returns.

The file system module implements the file API by mapping bytes of the file to disk sectors. For each file the file system creates a record of the name of the file and the disk sectors in which it has stored the file. The file system also stores this record on the disk. When the computer restarts, the file system must somehow discover the place where it left these records so that it can again find the files. A typical procedure for name discovery is for the file system to reserve one, well-known, disk sector such as sector number 1, and use that well-known disk sector as a toehold to locate the sectors where it left the rest of the file system information. A detailed description of the UNIX file system API and its implementation is in Section 2.5.

One might wonder why the file API supports OPEN and CLOSE in addition to READ and WRITE; after all, one could ask the programmer to pass the file name and a file position offset on each READ and WRITE call. The reason is that the OPEN and CLOSE procedures mark the beginning and the end of a sequence of related READ and WRITE operations so that the file system knows which reads and writes belong together as a group. There are several good reasons for grouping and for the use of a temporary file name within the grouping. Originally, performance and resource management concerns motivated the introduction of OPEN and CLOSE, but later implementations of the interface exploited the existence of OPEN and CLOSE to provide clean semantics under concurrent file access and failures.

Early file systems introduced OPEN to amortize the cost of resolving a file name. A file name is a path name that may contain several components. By resolving the file

name once on OPEN and giving the result a simple name, READ and WRITE avoid having to resolve the name on each invocation. Similarly, OPEN amortizes the cost of checking whether the user has the appropriate permissions to use the file.

CLOSE was introduced to simplify resource management: when an application invokes CLOSE, the file system knows that the application doesn't need the resources (e.g., the cursor) that the file system maintains internally. Even if a second application removes a file before a first application is finished reading and writing the file, the file system can implement READ and WRITE procedures for the first application sensibly (for example, discard the contents of the file only after everyone that OPENed the file has called CLOSE).

More recent file systems use OPEN and CLOSE to mark the beginning and end of an *atomic* action. The file system can treat all intervening READ and WRITE calls as a single indivisible operation, even in the face of concurrent access to the file or a system crash after some but not all of the WRITEs have completed. Two opportunities ensue:

1. The file system can use the OPEN and CLOSE operations to coordinate concurrent access to a file: if one program has a file open and another program tries to OPEN that same file, the file system can make the second program wait until the first one has CLOSEd the file. This coordination is an example of *before-or-after atomicity*, a topic that Section 5.2.4 explores in depth.

2. If the file system crashes (for example, because of a power failure) before the application CLOSEs the file, none of the WRITEs will be in the file when the system comes back up. If it crashes after the application CLOSEd the file, all of the WRITEs will be in the file. Not all file systems provide this guarantee, known as *all-or-nothing atomicity*, since it is not easy to implement correctly and efficiently, as Chapter 9 [on-line] explains.

There is a cost to the OPEN/CLOSE model: the file system must maintain per-client state in the form of the resolved file name and the cursor(s). It is possible to design a completely stateless file interface. An example is the Network File System, described in Section 4.5.

The file is such a convenient memory abstraction that in some systems (for example, the UNIX system and its derivatives) *every* input/output device in a computer system provides a file interface (see Figure 2.19). In such systems, files not only are an abstraction for non-volatile memories (e.g., magnetic disks), but they are also a convenient interface to the keyboard device, the display, communication links, and so on. In such systems, each I/O device has a name in the file naming scheme. A program OPENs the keyboard device, READs bytes from the keyboard device, and then CLOSEs the keyboard device, without having to know any details about the keyboard management procedure, what type of keyboard it is, and the like. Similarly, to interact with the display, a program can OPEN the display device, WRITE to it, and CLOSE it. The program need not know any details about the display. In accordance with the ***principle of least astonishment***, each device management procedure provides some reasonable interpretation for every file system method. The pseudocode of Figure 2.18 exemplifies the benefit of this kind of design uniformity.

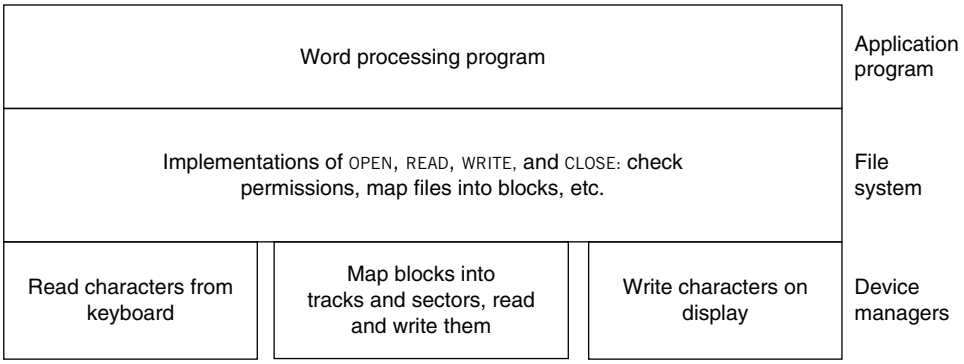


FIGURE 2.19

Using the file abstraction and layering to integrate different kinds of input and output devices. The file system acts as an intermediary that provides a uniform, abstract interface, and the various device managers are programs that translate that abstract interface into the operational requirements for different devices.

One feature of such a uniform interface is that in many situations one can, by simply rebinding the name, replace an I/O device with a file, or vice versa, without modifying the application program in any way. This use of naming in support of modularity is especially helpful when debugging an application program. For example, one can easily test a program that expects keyboard input by slipping a file filled with text in the place of the keyboard device. Because of such examples, the file system abstraction has proven to be very successful.

2.4 LOOKING BACK AND AHEAD

This chapter has developed several ideas and concepts that provide useful background for the study of computer system design. First, it described the three major abstractions used in designing computer systems—memory, interpreters, and communication links. Then it presented a model of how names are used to glue together modules based on those abstractions to create useful systems. Finally, it described some parts of a typical modern layered computer system in terms of the three major abstractions. With this background, we are now prepared to undertake a series of more in-depth discussions of specific computer system design topics. The first such in-depth discussion, in Chapter 3, is of the several engineering problems surrounding the use of names. Each of the remaining chapters undertakes a similar in-depth discussion of a different system design topic.

Before moving on to those in-depth discussions, the last section of this chapter is a case study of how abstraction, naming, and layers appear in practice. The case study uses those three concepts to describe the UNIX system.

2.5 CASE STUDY: UNIX® FILE SYSTEM LAYERING AND NAMING

The UNIX family of operating systems can trace its lineage back to the UNIX operating system that was developed by Bell Telephone Laboratories for the Digital Equipment Corporation PDP line of minicomputers in the late 1960s and early 1970s [Suggestions for Further Reading 2.2], and before that to the Multics* operating system in the early 1960s [Suggestions for Further Reading 1.7.5 and 3.1.4]. Today there are many flavors of UNIX systems with complex historical relationships; a few examples include GNU/Linux, versions of GNU/Linux distributed by different organizations (e.g., Red Hat, Ubuntu), Darwin (a UNIX operating system that is part of Apple's operating system Mac OS X), and several flavors of BSD operating systems. Some of these are directly derived from the early UNIX operating system; others provide similar interfaces but have been implemented from scratch. Some are the result of an effort by a small group of programmers, and others are the result of an effort by many. In the latter case, it is even unclear how to exactly name the operating system because substantial parts come from different teams.† The collective result of all these efforts is that operating systems of the UNIX family run on a wide range of computers, including personal computers, server computers, parallel computers, and embedded computers. Most of the UNIX interface is an official standard,‡ and non-UNIX operating systems often support this standard too. Because the source code of some versions is available to the public, one can easily study the UNIX system.

This case study examines the various ways in which the UNIX file system uses names in its design. In the course of examining how it implements its naming scheme, we will also incidentally get a first-level overview of how the UNIX file system is organized.

2.5.1 Application Programming Interface for the UNIX File System

A program can create a file with a user-chosen name, read and write the file's content, and set and get a file's metadata. Example metadata include the time of last modification, the user ID of the file's owner, and access permissions for other users. (For a full discussion of metadata see Section 3.1.2.) To organize their files, users can group them in directories with user-chosen names, creating a naming network. Users can also graft a naming network stored on a storage device onto an existing naming network, allowing naming networks for different devices to be incorporated into a single large nam-

*The name UNIX evolved from Unics, which was a word joke on Multics.

†We use "Linux" for the Linux kernel, while we use "GNU/Linux" for the complete system, recognizing that this naming convention is not perfect either because there are pieces of the system that are neither GNU software nor part of the kernel (e.g., the X Window System; see Sidebar 4.4).

‡POSIX® (Portable Operating System Interface), Federal Information Processing Standards (FIPS) 151-2. FIPS 151-2 adopts ISO/IEC 9945-1: 2003 (IEEE Std. 1003.1: 2001) Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application: Program Interface (API) [C Language].

Table 2.1 UNIX File System Application Programming Interface

Procedure	Brief Description
OPEN (*name, flags, mode*)	Open file *name*. If the file doesn't exist and *flags* is set, create file with permissions *mode*. Set the file cursor to 0. Returns a file descriptor.
READ (*fd, buf, n*)	Read *n* bytes from the file at the current cursor and increase the cursor by the number of bytes read.
WRITE (*fd, buf, n*)	Write *n* bytes at the current cursor and increase the cursor by the bytes written.
SEEK (*fd, offset, whence*)	Set the cursor to *offset* bytes from beginning, end, or current position.
CLOSE (*fd*)	Delete file descriptor. If this is the last reference to the file, delete the file.
FSYNC (*fd*)	Make all changes to the file durable.
STAT (*name*)	Read metadata of file.
CHMOD, CHOWN, etc.	Various procedures to set specific metadata.
RENAME (*from_name, to_name*)	Change name from *from_name* to *to_name*
LINK (*name, link_name*)	Create a hard link *link_name* to the file *name*.
UNLINK (*name*)	Remove *name* from its directory. If *name* is the last name for a file, remove file.
SYMLINK (*name, link_name*)	Create a symbolic name *link_name* for the file *name*.
MKDIR (*name*)	Create a new directory named *name*.
CHDIR (*name*)	Change current working directory to *name*.
CHROOT (*name*)	Change the default root directory to *name*.
MOUNT (*name, device*)	Graft the file system on *device* onto the name space at *name*.
UNMOUNT (*name*)	Unmount the file system at *name*.

ing network. To support these operations, the UNIX file system provides the application programming interface (API) shown in Table 2.1.

To tackle the problem of implementing this API, the UNIX file system employs a divide-and-conquer strategy. The UNIX file system makes use of several hidden layers of machine-oriented names (that is, addresses), one on top of another, to implement files. It then applies the UNIX durable object naming scheme to map user-friendly names to these files. Table 2.2 illustrates this structure.

In the rest of this section we work our way up from the bottom layer of Table 2.2 to the top layer, proceeding from the lowest layer of the system up toward

Table 2.2 The Naming Layers of the UNIX File System

Layer	Purpose	
Symbolic link layer	Integrate multiple file systems with symbolic links.	↑ user-oriented names
Absolute path name layer	Provide a root for the naming hierarchies.	
Path name layer	Organize files into naming hierarchies.	↓
File name layer	Provide human-oriented names for files.	machine-user interface
Inode number layer	Provide machine-oriented names for files.	↑ machine-oriented names
File layer	Organize blocks into files.	
Block layer	Identify disk blocks.	↓

the user. This description corresponds closely to the implementation of Version 6 of the UNIX system, which dates back to the early 1970s. Version 6 is well documented [Suggestions for Further Reading 2.2.2] and captures the important ideas that are found in many modern UNIX file systems, but modern versions are more complex; they provide better robustness and handle large files, many files, and so on, more efficiently. In a few places we will point out some of these differences, but the reader is encouraged to consult papers in the file system literature to find out how modern UNIX file systems work and are evolving.

2.5.2 The Block Layer

At the bottom layer the UNIX file system names some physical device such as a magnetic disk, flash disk, or magnetic tape that can store data durably. The storage on such a device is divided into fixed-size units, called *blocks*. For a magnetic disk (see Sidebar 2.2), a block corresponds to a small number of disk sectors. A block is the smallest allocation unit of disk space, and its size is a trade-off between several goals. A small block reduces the amount of disk wasted for small files; if many files are smaller than 4 kilobytes, a 16-kilobyte block size wastes space. On the other hand, a very small block size may incur large data structures to keep track of free and allocated blocks. In addition, there are performance considerations that impact the block size, some of which we discuss in Chapter 6. In version 6, the UNIX file system used 512-byte blocks, but modern UNIX file systems often use 8-kilobyte blocks.

The names of these blocks are numbers, which typically correspond to the offset of the block from the beginning of the device. In the bottom naming layer, a storage

device can be viewed as a context that binds block numbers to physical blocks. The name-mapping algorithm for a block device is simple: it takes as input a block number and returns the block. Actually, we don't really want the block itself—that would be a pile of iron oxide. What we want is the *contents* of the block, so the algorithm actually implements a fixed mapping between block name and block contents. If we represent the storage device as a linear array of blocks, then the following code fragment implements the name-mapping algorithm:

```
procedure BLOCK_NUMBER_TO_BLOCK (integer b) returns block
    return device[b]
```

In this simple algorithm the variable name *device* refers to some particular physical device. In many devices the mapping is more complicated. For example, a hard drive might keep a set of spare blocks at the end and rebind the block numbers of any blocks that go bad to spares. The hard drive may itself be implemented in layers, as will be seen in Section 8.5.4 [on-line]. The value returned by BLOCK_NUMBER_TO_BLOCK is the contents of block *b*.

Name discovery: The names of blocks are integers from a compact set, but the block layer must keep track of which blocks are in use and which are available for assignment. As we will see, the file system in general has a need for a description of the layout of the file system on disk. As an anchor for this information, the UNIX file system starts with a ***super block***, which has a well-known name (e.g., 1). The super block contains, for example, the size of the file system's disk in blocks. (Block 0 typically stores a small program that starts the operating system; see Sidebar 5.3.)

Different implementations of the UNIX file system use different representations for the list of free blocks. The version 6 implementation keeps a list of block numbers of unused blocks in a linked list that is stored in some of the unused blocks. The block number of the first block of this list is stored in the super block. A call to allocate a block leads to a procedure in the block layer that searches the list array for a free block, removes it from the list, and returns that block's block number.

Modern UNIX file systems often use a bitmap for keeping track of free blocks. Bit *i* in the bitmap records whether block *i* is free or allocated. The bitmap itself is stored at a well-known location on the disk (e.g., right after the super block). Figure 2.20 shows a possible disk layout for a simple file system. It starts with the super block, followed by a bitmap that records which disk blocks are in use. After the bitmap comes the inode table, which has one entry for each file (as explained next), followed by blocks that are either free or allocated to some file. The super block contains the size of the bitmap and inode table in blocks.

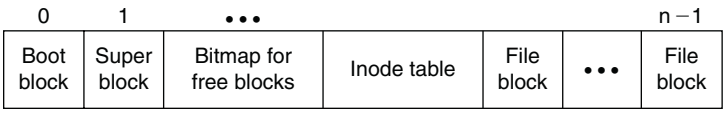


FIGURE 2.20
Possible disk layout for a simple file system.

2.5.3 The File Layer

Users need to store items that are larger than one block in size and that may grow or shrink over time. To support such items, the UNIX file system introduces a next naming layer for *files*. A file is a linear array of bytes of arbitrary length. The file system needs to record in some way which blocks belong to each file. To support this requirement, the UNIX file system creates an index node, or *inode* for short, as a container for metadata about the file. Our initial declaration of an inode is:

```
structure inode
    integer block_numbers[N]    // the numbers of the blocks that constitute the file
    integer size                // the size of the file in bytes
```

The inode for a file is thus a context in which the various blocks of the file are named by integer block numbers. With this structure, a simplified name-mapping algorithm for resolving the name of a block in a file is as follows:

```
procedure INDEX_TO_BLOCK_NUMBER (inode instance i, integer index) returns integer
    return i.block_numbers[index]
```

The version 6 UNIX file system uses this algorithm for small files, which are limited to $N = 8$ blocks. For large files, version 6 uses a more sophisticated algorithm for mapping the *index*-th block of an inode to a block number. The first seven entries in *i.block_numbers* are ***indirect blocks***. Indirect blocks do not contain data, but block numbers. For example, with a block size of 512 bytes and an *index* of 2 bytes (as in Version 6), an indirect block can contain 256 2-byte block numbers. The eighth entry is a doubly indirect block (blocks that contain block numbers of indirect blocks). This design with indirect and doubly indirect blocks allows for $(N - 1) \times 256 + 1 \times 256 \times 256 = 67{,}329$ blocks when $N = 8$, about 32 megabytes.* Problem set *1* explores some design trade-offs to allow the file system to support large files. Some modern UNIX file systems use different representations or more sophisticated data structures, such as B+ trees, to implement files.

The UNIX file system allows users to name any particular byte in a file by layering the previous two naming schemes and specifying the byte number as an offset from the beginning of the file:

```
1    procedure INODE_TO_BLOCK (integer offset, inode instance i) returns block
2        o ← offset / BLOCKSIZE
3        b ← INDEX_TO_BLOCK_NUMBER (i, o)
4        return BLOCK_NUMBER_TO_BLOCK (b)
```

The value returned is the entire block that holds the value of the byte at *offset*. Version 6 used for *offset* a 3-byte number, which limits the maximum file size to 2^{24} bytes. Modern UNIX file systems use a 64-bit number. The procedure returns the

*The implementation of Version 6, however, restricts the maximum number of blocks per file to 2^{15}.

entire block that contains the named byte. As we will see in Section 2.5.11, READ uses this procedure to return the requested bytes.

2.5.4 The Inode Number Layer

Instead of passing inodes themselves around, it would be more convenient to name them and pass their names around. To support this feature, the UNIX file system provides another naming layer that names inodes by an inode number. A convenient way to implement this naming layer is to employ a table that directly contains all inodes, indexed by inode number. Here is the naming algorithm:

```
1    procedure INODE_NUMBER_TO_INODE (integer inode_number) returns inode
2        return inode_table[inode_number]
```

where *inode_table* is an object that is stored at a fixed location on the storage device (e.g., at the beginning). The name-mapping algorithm for *inode_table* just returns the starting block number of the table.

Name discovery: inode numbers, like disk block numbers, are a compact set of integers, and again the inode number layer must keep track of which inode numbers are in use and which are free to be assigned. As with block number assignment, different implementations use various representations for a list of free inodes and provide calls to allocate and deallocate inodes. In the simplest implementation, the inode contains a field recording whether or not it is free.

By putting these three layers together, we obtain the following procedure:

```
1    procedure INODE_NUMBER_TO_BLOCK (integer offset, integer inode_number)
2                                                    returns block
3        inode instance i ← INODE_NUMBER_TO_INODE (inode_number)
4        o ← offset / BLOCKSIZE
5        b ← INDEX_TO_BLOCK_NUMBER (i, o)
6        return BLOCK_NUMBER_TO_BLOCK (b)
```

This procedure returns the block that contains the byte at *offset* in the file named by *inode_number*. This procedure traverses three layers of naming. There are numbers for storage blocks, numbered indexes for blocks belonging to an inode, and numbers for inodes.

2.5.5 The File Name Layer

Numbers are convenient names for use by a computer (numbers can be stored in fixed-length fields that simplify storage allocation) but are inconvenient names for use by people (numbers have little mnemonic value). In addition, block and inode numbers specify a location, so if it becomes necessary to rearrange the physical storage, the numbers must change, which is again inconvenient for people. The UNIX file system deals with this problem by inserting a naming layer whose sole purpose is

File name	Inode number
program	10
paper	12

FIGURE 2.21

A directory.

to hide the metadata of file management. Above this layer is a user-friendly naming scheme for durable objects—files and input/output devices. This naming scheme again has several layers. The most visible component of the durable object naming scheme is the *directory*. In the UNIX file system, a directory is a context containing a set of bindings between character-string names and inode numbers.

To create a file, the UNIX file system allocates an inode, initializes its metadata, and binds the proposed name to that inode in some directory. As the file is written, the file system allocates blocks to the inode.

By default, the UNIX file system adds the file to the current working directory. The current working directory is a context reference to the directory in which the active application is working. The form of the context reference is just another inode number. If *wd* is the name of the state variable that contains the working directory for a running program (called a *process* in the UNIX system), one can look up the inode number of the just-created file by supplying *wd* as the second argument to a procedure such as:

```
procedure NAME_TO_INODE_NUMBER (character string filename, integer dir) returns integer
    return LOOKUP (filename, dir)
```

The procedure CHDIR, whose implementation we describe later, allows a process to set *wd*.

To represent a directory, the UNIX file system reuses the mechanisms developed so far: it represents directories as files. By convention, a file that represents a directory contains a table that maps file names to inode numbers. For example, Figure 2.21 is a directory with two file names ("program" and "paper"), which are mapped to inode numbers 10 and 12, respectively. In Version 6, the maximum length of a name is 14 bytes, and the entries in the table have a fixed length of 16 bytes (14 for the name and 2 for the inode number). Modern UNIX file systems allow for variable-length names, and the table representation is more sophisticated.

To record whether an inode is for a directory or a file, the UNIX file system extends the inode with a type field:

```
structure inode
    integer block_numbers[N]    // the numbers of the blocks that constitute the file
    integer size                // the size of the file in bytes
    integer type                // type of file: regular file, directory, . . .
```

MKDIR creates a zero-length file (directory) and sets *type* to DIRECTORY. Extensions introduced later will add additional values for *type*.

With this representation of directories and inodes, LOOKUP is as follows:

```
1    procedure LOOKUP (character string filename, integer dir) returns integer
2        block instance b
3        inode instance i ← INODE_NUMBER_TO_INODE (dir)
4        if i.type ≠ DIRECTORY then return FAILURE
5        for offset from 0 to i.size – 1 do
6            b ← INODE_NUMBER_TO_BLOCK (offset, dir)
7            if STRING_MATCH (filename, b) then
8                return INODE_NUMBER (filename, b)
9            offset ← offset + BLOCKSIZE
10       return FAILURE
```

LOOKUP reads the blocks that contain the data for the directory *dir* and searches for the string *filename* in the directory's data. It computes the block number for the first block of the directory (line *6*) and the procedure STRING_MATCH (no code shown) searches that block for an entry for the name *filename*. If there is an entry, INODE_NUMBER (no code shown) returns the inode number in the entry (line *8*). If there is no entry, LOOKUP computes the block number for the second block, and so on, until all blocks of the directory have been searched. If none of the blocks contain an entry for *filename*, LOOKUP returns an error (line *10*). As an example, an invocation of LOOKUP ("program", *dir*), where *dir* is the inode number for the directory of Figure 2.21, would return the inode number 10.

2.5.6 The Path Name Layer

Having all files in a single directory makes it hard for users to keep track of large numbers of files. Enumerating the contents of a large directory would generate a long list that is organized simply (e.g., alphabetically) at best. To allow arbitrary groupings of user files, the UNIX file system permits users to create named directories.

A directory can be named just like a file, but the user also needs a way of naming the files in that directory. The solution is to add some structure to file names: for example, "projects/paper", in which "projects" names a directory and "paper" names a file in that directory. Structured names such as these are examples of path names. The UNIX file system uses a virgule (forward slash) as a separator of the components of a path name; other systems choose different separator characters such as period, back slash, or colon. With these tools, users can create a hierarchy of directories and files.

The name-resolving algorithm for path names can be implemented by layering a recursive procedure over the previous directory lookup procedure:

```
1    procedure PATH_TO_INODE_NUMBER (character string path, integer dir) returns integer
2        if (PLAIN_NAME (path)) return NAME_TO_INODE_NUMBER (path, dir)
3        else
4            dir ← LOOKUP (FIRST (path), dir)
5            path ← REST (path)
6            return PATH_TO_INODE_NUMBER (path, dir)
```

The function PLAIN_NAME (*path*) scans its argument for the UNIX standard path name separator (forward slash) and returns TRUE if it does not find one. If there is no

separator, the program resolves the simple name to an inode number in the requested directory (line *2*). If there is a separator in *path*, the program takes it to be a path name and goes to work on it (lines *4* through *6*). The function FIRST peels off the first component name from the path, and REST returns the remainder of the path name. Thus, for example, the call PATH_TO_NAME ("projects/paper", *wd*) results in the recursive call PATH_TO_NAME ("paper", *dir*), where *dir* is the inode number for the directory "projects".

With path names, one often has to type names with many components. To address this annoyance, the UNIX file system supports a change directory procedure, CHDIR, allowing a process to set its working directory:

```
procedure CHDIR (path character string)
    wd ← PATH_TO_INODE_NUMBER (path, wd)
```

When a process starts, it inherits the working directory from the parent process that created this process.

2.5.7 Links

To refer to files in directories other than the current working directory still requires typing long names. For example, while we are working in the directory "projects"—after calling CHDIR ("projects")—we might have to refer often to the file "Mail/inbox/new-assignment". To address this annoyance, the UNIX file system supports synonyms known as *links*. In the example, we might want to create a link for this file in the current working directory, "projects". Invoking the LINK procedure with the following arguments:

```
LINK ("Mail/inbox/new-assignment", "assignment")
```

makes "assignment" a synonym for "Mail/inbox/new-assignment" in "projects", if "assignment" doesn't exist yet. (If it does, LINK will return an error saying "assignment" already exists.) With links, the directory hierarchy turns from a strict hierarchy into a directed graph. (The UNIX file system allows links only to files, not to directories, so the graph is not only directed but acyclic. We will see why in a moment.)

The UNIX file system implements links simply as bindings in different contexts that map different file names to the same inode number; thus, links don't require any extension to the naming scheme developed so far. For example, if the inode number for "new-assignment" is 481, then the directory "Mail/inbox" contains an entry {"new-assignment", 481} and after the above command is executed the directory "projects" contains an entry {"assignment", 481}. In UNIX system jargon, "projects/assignment" is now linked to "Mail/inbox/new-assignment".

When a file is no longer needed, a process can remove a file using UNLINK (*filename*), indicating to the file system that the name *filename* is no longer in use. UNLINK removes the binding of *filename* to its inode number from the directory that contains *filename*. The file system also puts *filename*'s inode and the blocks of *filename*'s inode on the free list if this binding is the last one containing the inode's number.

Before we added links, a file was bound to a name in only one directory, so if a process asks to delete the name from that directory, the file system can also delete the file. But now that links have been added, when a process asks to delete a name, there may still be names in other directories bound to the file, in which case the file shouldn't be deleted. This raises the question, when should a file be deleted? The UNIX file system deletes a file when a process removes the last binding for a file. The UNIX file system implements this policy by keeping a reference count in the inode:

```
structure inode
    integer block_numbers[N]
    integer size
    integer type
    integer refcnt
```

Whenever it makes a binding to an inode, the file system increases the reference count of that inode. To delete a file, the UNIX file system provides an UNLINK(*filename*) procedure, which deletes the binding specified by *filename*. At the same time the file system decreases the reference count in the corresponding inode by one. If the decrease causes the reference count to go to zero, that means there are no more bindings to this inode, so the file system can free the inode and its corresponding blocks. For example, UNLINK ("Mail/inbox/new-assignment") removes the directory entry "new-assignment" in the directory "Mail/inbox", but not "assignment", because after the unlink the *refcnt* in inode 481 will be 1. Only after calling UNLINK ("assignment") will the inode 481 and its blocks be freed.

Using reference counts works only if there are no cycles in the naming graph. To ensure that the UNIX naming network is a directed graph without cycles, the UNIX file system forbids links to directories. To see why cycles are avoided, consider a directory "a", which contains a directory "b". If a program invokes LINK ("a/b/c", "a") in the directory that contains "a", then the system would return an error and not perform the operation. If the system had performed this operation, it would have created a cycle from "c" to "a" and would have increased the reference count in the inode of "a" by one. If a program then invokes UNLINK ("a"), the name "a" is removed, but the inode and the blocks of "a" wouldn't be removed because the reference count in the inode of "a" is still positive (because of the link from "c" to "a"). But once the name "a" would be removed, a user would no longer be able to name the directory "a" and wouldn't be able to remove it either. In that case, the directory "a" and its subdirectories would be disconnected from the naming graph, but the system would not remove it because the reference count in the inode of "a" is still positive. It is possible to detect this situation, for example by using garbage collection, but it is expensive to do so. Instead, the designers chose a simpler solution: don't allow links to directories, which rules out the possibility of cycles.

There are two special cases, however. First, by default each directory contains a link to itself; the UNIX file system reserves the string "." (a single dot) for this purpose. The name "." thus allows a process to name the current directory without knowing

which the directory it is. When a directory is created, the directory's inode has a reference count of two: one for the inode of the directory and one for the link ".", because it points to itself. Because "." introduces a cycle of length 0, there is no risk that part of the naming network will become disconnected when removing a directory. When unlinking a directory, the file system just decreases the reference count of the directory's inode by 2.

Second, by default, each directory also contains a link to a parent directory; the file system reserves the string ".." (two consecutive dots) for this purpose. The name ".." allows a process to name a parent directory and, for example, move up the file hierarchy by invoking CHDIR ("..."). The link doesn't create problems. Only when a directory has no other entries than "." and ".." can it be removed. If a user wants to remove a directory "a", which contains a directory "b", then the file system refuses to do so until the user first has removed "b". This rule ensures that the naming network cannot become disconnected.

2.5.8 Renaming

Using LINK and UNLINK, Version 6 implemented RENAME (*from_name*, *to_name*) as follows:

```
1   UNLINK (to_name)
2   LINK (from_name, to_name)
3   UNLINK (from_name)
```

This implementation, however, has an undesirable property. Programs often use RENAME to change a working copy of a file into the official version; for example, a user may be editing a file "x". The text editor actually makes all changes to a temporary file "#x". When the user saves the file, the editor renames the temporary file "#x" to "x".

The problem with implementing RENAME using LINK and UNLINK is that if the computer fails between steps 1 and 2 and then restarts, the name *to_name* ("x" in this case) will be lost, which is likely to surprise the user, who is unlikely to know that the file still exists but under the name "#x". What is really needed is that "#x" be renamed to "x" in a single, atomic operation, but that requires atomic actions, which are the topic of Chapter 9 [on-line].

Without atomic actions, it is possible to implement the following slightly weaker specification for RENAME: if *to_name* already exists, an instance of *to_name* will always exist, even if the system should fail in the middle of RENAME. This specification is good enough for the editor to do the right thing and is what modern versions provide.

Modern versions implement this specification in essence as follows:

```
1   LINK (from_name, to_name)
2   UNLINK (from_name)
```

Because one cannot link to a name that already exists, RENAME implements the effects of these two calls by manipulating the file system structures directly. RENAME first changes the inode number in the directory entry for *to_name* to the inode number for

from_name on disk. Then, RENAME removes the directory entry for *from_name*. If the file system fails between these two steps, then on recovery the file system must increase the reference count in *from_name*'s inode because both *from_name* and *to_name* are pointing to the inode. This implementation ensures that if *to_name* exists before the call to RENAME, it will continue to exist, even if the computer fails during RENAME.

2.5.9 The Absolute Path Name Layer

The UNIX system provides each user with a personal directory, called a user's *home directory*. When a user logs on to a UNIX system, it starts a command interpreter (known as the *shell*) through which a user can interact with the system. The shell starts with the working directory (*wd*) set to the inode number of the user's home directory. With the above procedures, users can create personal directory trees to organize the files in their home directory.

But having several personal directory trees does not allow one user to share files with another. To do that, one user needs a way of referring to the names of files that belong to another user. The easiest way to accomplish that is to bind a name for each user to that user's top-level directory, in some context that is available to every user. But then there is a requirement to name this systemwide context. Typically, there are needs for other systemwide contexts, such as a directory containing shared program libraries. To address these needs with a minimum of additional mechanisms, the file system provides a universal context, known as the *root* directory. The root directory contains bindings for the directory of users, the directory containing program libraries, and any other widely shared directories. The result is that all files of the system are integrated into a single directory tree (with restricted cross-links) based on the root.

This design leaves a name discovery question: how can a user name the root directory? Recall that name lookup requires a context reference—the name of a directory inode—and until now that directory inode has been supplied by the working directory state variable. To implement the root, the file system simply declares inode number 1 to be the inode for the root directory. This *well-known name* can then be used by any user as the starting context in which to look up the name of a shared context, or another user (or even to look up one's own name, to set the working directory when logging in).

The file system actually provides two ways to refer to things in the root directory. Starting from any directory in the system, one can use the name ".." to name that directory's parent, "../.." to name the directory above that, and so on until the root directory is reached. A user can tell that the root directory is reached, because ".." in the root directory names the root directory. That is, in the root directory, both "." and ".." are links to the root directory. The other way is with absolute path names, which in the UNIX file system are names that start with a "/", for example, "/Alice/Mail/inbox/new-assignment".

To support absolute path names as well as relative path names, we need one more layer in the naming scheme:

```
1    procedure GENERALPATH_TO_INODE_NUMBER (character string path) returns integer
2        if (path[0] = "/") return PATH_TO_INODE_NUMBER(path, 1)
3        else return PATH_TO_INODE_NUMBER(path, wd)
```

At this point we have completed a naming scheme that allows us to name and share durable storage on a single disk. For example, to find the blocks corresponding to the file "/programs/pong.c" with the information in Figure 2.22, we start by finding the inode table, which starts at a block number (block 4 in our example) stored in the super block (not shown in this figure, but see Figure 2.20). From there we locate the root inode (which is known to be inode number 1). The root inode contains the block numbers that in turn contain the blocks of the root directory; in the figure the root starts in block number 14. Block 14 lists the entries in the root directory: "programs" is named by inode number 7. The inode table says that data for inode number 7 starts in block number 23, which contains the contents of the "programs" directory. The file "pong.c" is named by inode number 9. Referring once more to the inode table, to see where inode 9 is stored, we see that the data corresponding to inode 9 starts in block number 61. In short, directories and files are carefully laid out so that all information can be found by starting from the well-known location of the root inode.

The default root directory in Version 6 is inode 1. Version 7 added a call, CHROOT, to change the root directory for a process. For example, a Web server can be run in the corner of the UNIX name space by changing its root directory to, for example, "/tmp". After this call, the root directory for the Web server corresponds to the inode number of the directory "/tmp" and ".." in "/tmp" is a link to "/tmp". Thus, the server can name only directories and files below "/tmp".

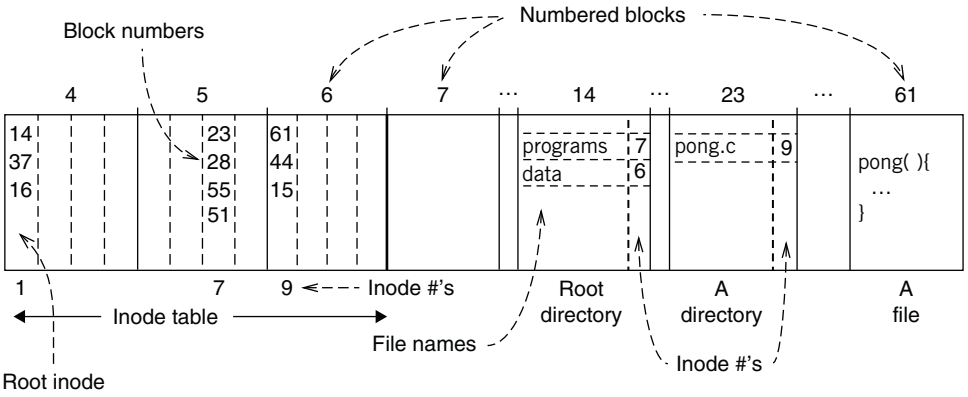


FIGURE 2.22

Example disk layout for a UNIX file system, refining Figure 2.20 by focusing on the inode table and data blocks. The inode table is a group of contiguous blocks starting at a well-known address, found in the super block (not shown). In this example, blocks 4, 5, and 6 contain the inode table, while blocks 7–61 contain directories and files. The root inode is by convention the well-known inode #1. Typically, inodes are smaller than a block, so in this example there are four inodes in each block. Blocks #14, #37, and #16 constitute the root directory, while block #23 is the first of four blocks of the directory named "/programs", and block #61 is the first block of the three-block file "/programs/pong.c".

2.5.10 The Symbolic Link Layer

To allow users to name files on other disks, the UNIX file system supports an operation to attach new disks to the name space. A user can choose the name under which each device is attached: for example, the procedure

```
MOUNT ("/dev/fd1", "/flash")
```

grafts the directory tree stored on the physical device named "/dev/fd1" onto the directory "/flash". (This command demonstrates that each device also has a name in the same object name space we have been describing; the file corresponding to a device typically contains information about the device itself.) Typically mounts do not survive a shutdown: after a reboot, the user has to explicitly remount the devices. It is interesting to contrast the elegant UNIX approach with the DOS approach, in which devices are named by fixed one-character names (e.g., "C:").

The UNIX file system implements MOUNT by recording in the in-memory inode for "flash" that a file system has been mounted on it and keeps this inode in memory until at least the corresponding UNMOUNT. In memory, the system also records the device and the root inode number of the file system that has been mounted on it. In addition, it records in the in-memory version of the inode for "/dev/fd1" what its parent inode is.

The information for mount points is all recorded in volatile memory instead of on disk and doesn't survive a computer failure. After a failure, the system administrator or a program must invoke MOUNT again. Supporting MOUNT also requires a change to the file name layer: if LOOKUP runs into an inode on which a file system is mounted, it uses the root inode of the mounted file system for the lookup.

UNMOUNT undoes the mount.

With mounted file systems, synonyms become a more difficult problem because per mounted file system there is an address space of inode numbers. Every inode number has a default context: the disk on which it is located. Thus, there is no way for a directory entry on one disk to bind to an inode number on a different disk. This problem can be approached in several ways, two of which are: (1) make inodes unique across all disks or (2) create synonyms for files on other disks in a different way. The UNIX system chooses the second approach by using indirect names called *symbolic* or *soft* links, which bind a file name to another file name. Most systems use method (2) because of the complications that would be involved in trying to keep inode numbers universally unique, small in size, and fast to resolve.

Using the procedure SYMLINK, users can create synonyms for files in the same file system or for files in mounted file systems. The file system implements the procedure SYMLINK by allowing the type field of an inode to be a SYMLINK, which tells whether the blocks associated with the inode contain data or a path name:

```
structure inode
    integer block_numbers[N]
    integer size
    integer type        // Type of inode: regular file, directory, symbolic link, . . .
    integer refcnt
```

If the *type* field has value SYMLINK, then the data in the array *blocks*[*i*] actually contains the characters of a path name rather than a set of inode numbers.

Soft links can be implemented by layering them over GENERALPATH_TO_NODE_NUMBER:

```
1    procedure PATHNAME_TO_INODE (character string filename) returns inode
2        inode instance i
3        inode_number ← GENERALPATH_TO_INODE_NUMBER (filename)
4        i ← INODE_NUMBER_TO_INODE (inode_number)
5        if i.type = SYMBOLIC then
6            i = GENERALPATH_TO_INODE_NUMBER (COERCE_TO_STRING (i.block_numbers))
7        return i
```

The value returned by PATHNAME_TO_INODE is the contents of the inode for the file named by *filename*. The procedure first looks up the inode number for *filename*. Then, it looks up the the inode using INODE_NUMBER_TO_INODE. If the inode indicates that this file is a symbolic link, the procedure interprets the contents of data of the file as a path name and invokes GENERALPATH_TO_INODE_NUMBER again.

We now have two types of synonyms. A direct binding to an inode number is called a ***hard link***, to distinguish it from a soft link. Continuing an earlier example, a soft link to "Mail/inbox/new-assignment" would contain the string "Mail/inbox/new-assignment", rather than the inode number 481. A soft link is an example of an indirect name: it binds a name to another name in the same name space, while a hard link binds a name to an inode number, which is a name in a lower-layer name space. As a result, the soft link depends on the file name "Mail/inbox/new-assignment"; if the user changes the file's name or deletes the file, then "projects/assignment", the link, will end up as a dangling reference (Section 3.1.6 discusses dangling references). But because it links by name rather than by inode number, a soft link can point to a file on a different disk.

Recall that the UNIX system forbids cycles of hard links, so that it can use reference counts to detect when it is safe to reclaim the disk space for a file. However, you can still form cycles with soft links: a name deep down in the tree can, for example, name a directory high up in the tree. The resulting structure is no longer a directed acyclic graph, but a fully general naming network. Using soft links, a program can even invoke SYMLINK ("cycle", "cycle"), creating a synonym for a file name that doesn't have a file associated with it! If a process opens such a file, it will follow the link chain only a certain number of steps before reporting an error such as "Too many levels of soft links".

Soft links have another interesting behavior. Suppose that the working directory is "/Scholarly/programs/www" and that this working directory contains a symbolic link named "CSE499-web" to "/Scholarly/CSE499/www". The following calls

```
CHDIR ("CSE499-web")
CHDIR ("..")
```

leave the caller in "/Scholarly/CSE499" rather than back where the user started. The reason is that ".." is resolved in the new default context, "/Scholarly/CSE499/www", rather than what might have been the intended context, "/Scholarly/programs/www". This behavior may be desirable or not, but it is a direct consequence of the UNIX

Table 2.3 The UNIX Naming Layers, with Details of the Naming Scheme of Each Layer

Layer	Names	Values	Context	Name-Mapping Algorithm	
Symbolic link	Path names	Path names	The directory hierarchy	PATHNAME_TO_GENERAL_PATH	↑
Absolute path name	Absolute path names	Inode numbers	The root directory	GENERALPATH_TO_INODE_NUMBER	user-oriented names
Path name	Relative path names	Inode numbers	The working directory	PATH_TO_INODE_NUMBER	↓
File name	File names	Inode numbers	A directory	NAME_TO_INODE_NUMBER	machine-user interface
Inode number	Inode numbers	Inodes	The inode table	INODE_NUMBER_TO_INODE	↑
File	Index numbers	Block numbers	An inode	INDEX_TO_BLOCK_NUMBER	machine-oriented names
Block	Block numbers	Blocks	The disk drive	BLOCK_NUMBER_TO_BLOCK	↓

naming semantics; the Plan 9 system has a different plan,* which is also explored in exercises 3.2 and 3.3.

In summary, much of the power of the UNIX object naming scheme comes from its layers of naming. Table 2.3 reprises Table 2.2, this time showing the name, value, context, and pseudocode procedure used at each layer interface. (Although we have examined each of the layers in this table, the algorithms we have demonstrated have in some cases bridged across layers in ways not suggested by the table.) The general design technique has been to introduce for each problem another layer of naming, an application of the principle *decouple modules with indirection*.

2.5.11 Implementing the File System API

In the process of describing how the UNIX file system is structured, we saw how it implements CHDIR, MKDIR, LINK, UNLINK, RENAME, SYMLINK, MOUNT, and UNMOUNT. We complete the description of the file system API by describing the implementation of OPEN, READ, WRITE, and CLOSE. Before describing their implementation, we describe what features they must support.

*Rob Pike. Lexical File Names in Plan 9 or Getting Dot-Dot Right. *Proceedings of the 2000 USENIX Technical Conference* (2000), San Diego, pages 85–92.

The file system allows users to control who has access to their files. An owner of a file can specify with what permissions other users can make accesses to the file. For example, the owner may specify that other users have permission only to read a file but not to write it. OPEN must check whether the caller has the appropriate permissions. As a sophistication, a file can be owned by a group of users. Chapter 11 [on-line] discusses security in detail, so we will skip the details here.

The file system records timestamps that capture the date and time of the last access, last modification to a file, and last change to a file's inode. This information is important for programs such as incremental backup, which must determine which files have changed since the last time backup ran. The file system procedures must update these values. For example, READ updates last access time, WRITE updates last modification time and change time, and LINK updates last change time.

OPEN returns a short name for a file, called a file descriptor (*fd*), which READ, WRITE, and CLOSE use to name the file. Each process starts with three open files: "standard in" (file descriptor 0), "standard out" (file descriptor 1), and "standard error" (file descriptor 2). A file descriptor may name a keyboard device, a display device, or a file on disk; a program doesn't need to know. This setup allows a designer to develop a program without having to worry about where the program's input is coming from and where the program's output is going to; the program just reads from file descriptor 0 and writes to file descriptor 1.

Several processes can use a file concurrently (e.g., several processes might write to the display device). If several processes open the same file, their READ and WRITE operations have their own file cursor for that file. If one process opens a file, and then passes the file descriptor for that file to another process, then the two processes share the cursor of the file. This latter case is common because in the UNIX system when one process (the *parent*) starts another process (the *child*), the child inherits all open file descriptors from the parent. This design allows the parent and child, for instance, to share a common output file correctly. If the child writes to the output file, for example, after the parent has written to it, the output of the child appears after the output of the parent because they share the cursor.

If one process has a file open and another process removes the last name pointing to that file, the inode isn't freed until the first process calls CLOSE.

To support these features, the inode is extended as follows:

```
structure inode
    integer block_numbers[N]      // the number of blocks that constitute the file
    integer size                  // the size of the file in bytes
    integer type                  // type of file: regular file, directory, symbolic link
    integer refcnt                // count of the number of names for this inode
    integer userid                // the user ID that owns this inode
    integer groupid               // the group ID that owns this inode
    integer mode                  // inode's permissions
    integer atime                 // time of last access (READ, WRITE, . . . )
    integer mtime                 // time of last modification
    integer ctime                 // time of last change of inode
```

To implement OPEN, READ, WRITE, and CLOSE, the file system keeps in memory several tables: one file table (*file_table*) and for each process a file descriptor table (*fd_table*). The file table records information for the files that processes have open (i.e., files for which OPEN was successful, but for which CLOSE hasn't been called yet). For each open file, this information includes the inode number of the file, its file cursor, and a reference count recording how many processes have the file open. The file descriptor table records for each file descriptor the index into the file table. Because a file's cursor is stored in the *file_table* instead of the *fd_table*, children can share the cursor for an inherited file with their parent.

With this information, OPEN is implemented as follows:

```
1    procedure OPEN (character string filename, flags, mode)
2        inode_number ← PATH_TO_INODE_NUMBER (filename, wd)
3        if inode_number = FAILURE and flags = O_CREATE then      // Create the file?
4            inode_number ← CREATE (filename, mode)               // Yes, create it.
5        if inode_number = FAILURE then
6            return FAILURE
7        inode ← INODE_NUMBER_TO_INODE (inode_number)
8        if PERMITTED (inode, flags) then   // Does this user have the required permissions
9            file_index ← INSERT (file_table, inode_number)
10           fd ← FIND_UNUSED_ENTRY (fd_table)   // Find entry in file descriptor table
11           fd_table[fd] ← file_index           // Record file index for file descriptor
12           return fd                           // Return fd
13       else return FAILURE                     // No, return a failure
```

Line *2* finds the inode number for the file *filename*. If the file doesn't exist, but the caller wants to create the file as indicated by the flag O_CREATE (line *3*), OPEN calls CREATE, which allocates an inode, initializes it, and returns its inode number (line *4*). If the file doesn't exist (even after trying to create it), OPEN returns a value indicating a failure (line *6*). Line *7* locates the inode. Line *8* uses the information in the inode to check if the caller has permission to open the file; the check is described in detail in Section 11.6.3.4 [on-line]. If so, line *9* creates a new entry for the inode number in the file table and sets the entry's file cursor to zero and reference count to 1. Line *10* finds the first unused file descriptor, records its file index, and returns the file descriptor to the caller (lines *10* through *12*). Otherwise, it returns a value indicating a failure (line *13*).

If a process starts another process, the child process inherits the open file descriptors of the parent. That is, the information in every used entry in the parent's *fd_table* is copied to the same numbered entry in the child's *fd_table*. As a result, the parent and child entries in the *fd_table* will point to the same entry in the *file_table*, resulting in the cursor being shared between parent and child.

READ is implemented as follows:

```
1    procedure READ (fd, character array reference buf, n)
2        file_index ← fd_table[fd]
3        cursor ← file_table[file_index].cursor
4        inode ← INODE_NUMBER_TO_INODE (file_table[file_index].inode_number)
5        m = MINIMUM (inode.size – cursor, n)
6        atime of inode ← NOW ()
7        if m = 0 then return END_OF_FILE
8        for i from 0 to m – 1 do {
9            b ← INODE_NUMBER_TO_BLOCK (i, inode_number)
10           COPY (b, buf, MINIMUM (m – i, BLOCKSIZE))
11           i ← i + MINIMUM (m – i, BLOCKSIZE)
12       file_table[file_index].cursor ← cursor + m
13       return m
```

Lines *2* and *3* use the file index to find the cursor for the file. Line *4* locates the inode. Line *5* and *6* compute how many bytes READ can read and updates the last access time. If there are no bytes left in the file, READ returns a value indicating end of file. Lines *8* through *12* copy the bytes from the file's blocks into the caller's *buf*. Line *13* updates the cursor.

One could design a more sophisticated naming scheme for READ that, for example, allowed naming by keywords rather than by offsets. Database systems typically implement such naming schemes by representing the data as structured records that are indexed by keywords. But in order to keep its design simple, the UNIX file system restricts its representation of a file to a linear array of bytes.

The implementation of WRITE is similar to READ. The major differences are that it copies *buf* into the blocks of the inode, allocating new blocks as necessary, and that it updates the inode's *size* and *mtime*.

CLOSE frees the entry in the file descriptor table and decreases the reference count in entry in the file table. If no other processes are sharing this entry (i.e., the reference count has reached zero), it also frees the entry in the file table. If there are no other entries in the file table using this file and the reference count in the file's inode has reached zero (because another process unlinked it), then CLOSE frees the inode.

Like RENAME, some of these operations require several disk writes to complete. If the file system fails (e.g., because the power goes off) in the middle of one of the operations, then some of the disk writes may have completed and some may not. Such a failure can cause inconsistencies among the on-disk data structures. For example, the on-disk free list may show that a block is allocated, but no on-disk inode records that block in its index. If nothing is done about this inconsistency, then that block is effectively lost. Problem set *8* explores this problem and a simple, special-case solution. Chapter 9 [on-line] explores systematic solutions.

Version 6 (and all modern implementations) maintain an in-memory cache of recently used disk blocks. When the file system needs a block, it first checks the cache for the block. If the block is present, it uses the block from the cache; otherwise, it reads it from the storage device. With the cache, even if the file system needs to read a particular block several times, it reads that block from the storage device only once. Since reading from a disk device is often an expensive operation, the cache can improve the performance of the file system substantially. Chapter 6 discusses the implementation of caches in detail and how they can be used to improve the performance of a file system.

Similarly, to achieve high performance on operations that modify a file (e.g., WRITE), the file system will update the file's blocks in the cache, but will not force the file's modified inode and blocks to the storage device immediately. The file system delays the writes until later so that if a block is updated several times, it will write the block only once. Thus, it can coalesce many updates in one write (see Section 6.1.8).

If a process wants to ensure that the results of a write and inode changes are propagated to the device that stores the file system, it must call FSYNC; the UNIX specification requires that if an invocation of FSYNC for a file returns, all changes to the file must have been written to the storage device.

2.5.12 The Shell and Implied Contexts, Search Paths, and Name Discovery

Using the file system API, the UNIX system implements programs for users to manipulate files and name spaces. These programs include text editors (such as `ed`, `vi`, and `emacs`), `rm` (to remove a file), `ls` (to list a directory's content), `mkdir` (to make a new directory), `rmdir` (to remove a directory), `ln` (to make link names), `cd` (to change the working directory), and `find` (to search for a file in a directory tree).

One of the more interesting UNIX programs is its command interpreter, known as the "shell". The shell illustrates a number of other UNIX naming schemes. Say a user wants to compile the C source file named "x.c". The UNIX convention is to overload a file name by appending a suffix indicating the type of the file, such as ".c" for C source files. (A full discussion of overloading can be found in Section 3.1.2.) The user types this command to the shell:

```
CC X.C
```

This command consists of two names: the name of a program (the compiler "cc") and the name of a file containing source code ("x.c") for the compiler to compile. The first thing the shell must do is find the program we want to run, "cc". To do that, the UNIX command interpreter uses a default context reference contained in an environment variable named PATH. That environment variable contains a list of contexts (in this case directories) in which to perform a multiple lookup for the thing named "cc". Assuming the lookup is successful, the shell launches the program, calling it with the argument "x.c".

The first thing the compiler does is try to resolve the name "x.c". This time it uses a different default context reference: the working directory. Once the compilation

is underway, the file "x.c" may contain references to other named files, for example, statements such as

```
#include <stdio.h>
```

This statement tells the compiler to include all definitions in the file "stdio.h" in the file "x.c". To resolve "stdio.h", the compiler needs a context in which to resolve it. For this purpose, the compiler consults another variable (typically passed as an argument when invoking the compiler), which contains a default context to be used as a search path where include files may be found. The variables used by the shell and by the compiler each consist of a series of path names to be used as the basis for an ordered multiple lookup just as was described in Section 2.2.4.

Many other UNIX programs, such as the documentation package, `man`, also do multiple lookups for files using search paths found in environment variables.

The shell resolves names for commands using the PATH variable, but sometimes it is convenient to be able to say "I want to run the program located in the current working directory". For example, a user may be developing a new version of the C compiler, which is also called "cc". If the user types `"cc"`, the shell will look up the C compiler using the PATH variable and find the standard one instead of the new one in the current working directory.

For these cases, users can type the following command:

```
./cc x.c
```

which bypasses the PATH variable and invokes the program named "cc" in the current working directory (".").

Of course, the user could insert "." at the beginning of the PATH variable, so that all programs in the user's working directory will take precedence over the corresponding standard program. That practice, however, may create some surprises. Suppose "." is first entry in the PATH variable, and a user issues the following command sequence to the shell:

```
cd /usr/potluck
ls
```

intending to list the contents of the directory named `potluck`. If that directory contained a program named `ls` that did something different from the standard `ls` command, something surprising might happen (e.g., the program named `ls` could remove all private files)! For this reason, it is not a good idea to include names that are context-dependent, such as "." or ".." in a search path. It is better to include the absolute path name of the desired directory to the front of PATH.

Another command interpreter extension is that names can be descriptive rather than simple names. For example, the descriptive name "*.c", matches all file names that end with ".c". To provide this extension, the command interpreter transforms the single argument into a list of arguments (with the help of a more complicated lookup operation on the entries in the context) before it calls the specified command program. In the UNIX shell, users can use full-blown regular expressions in descriptive names.

As a final note, in practice, the UNIX object naming space has quite a bit of conventional structure. In particular, there are several directories with well-known names. For example, "/bin" names programs, "/etc" names configuration files, "/dev" names input/output devices, and "/usr" (rather than the root itself) names user directories. Over time these conventions have become so ingrained both in programmers' minds and in programs that much UNIX software will not install correctly, and a UNIX wizard will become badly confused, when confronted with a system that does not follow these conventions.

2.5.13 Suggestions for Further Reading

For a detailed description of a more modern UNIX operating system, see the book describing the BSD operating system [Suggestions for Further Reading 1.3.4]. A descendant of the original UNIX system is Plan 9 [Suggestions for Further Reading 3.2.2], which contains a number of novel naming abstractions, some of which are finding their way back into newer UNIX implementations. A rich literature exists describing file system implementations and their trade-offs. A good starting point are the papers on FFS [Suggestions for Further Reading 6.3.2], LFS [Suggestions for Further Reading 9.3.1], and soft updates [Suggestions for Further Reading 6.3.3].

EXERCISES

2.1 Ben Bitdiddle has accepted a job with the telephone company and has been asked to implement call forwarding. He has been pondering what to do if someone forwards calls to some number and then the owner of that number forwards calls to a third number. So far, Ben has thought of two possibilities for his implementation:

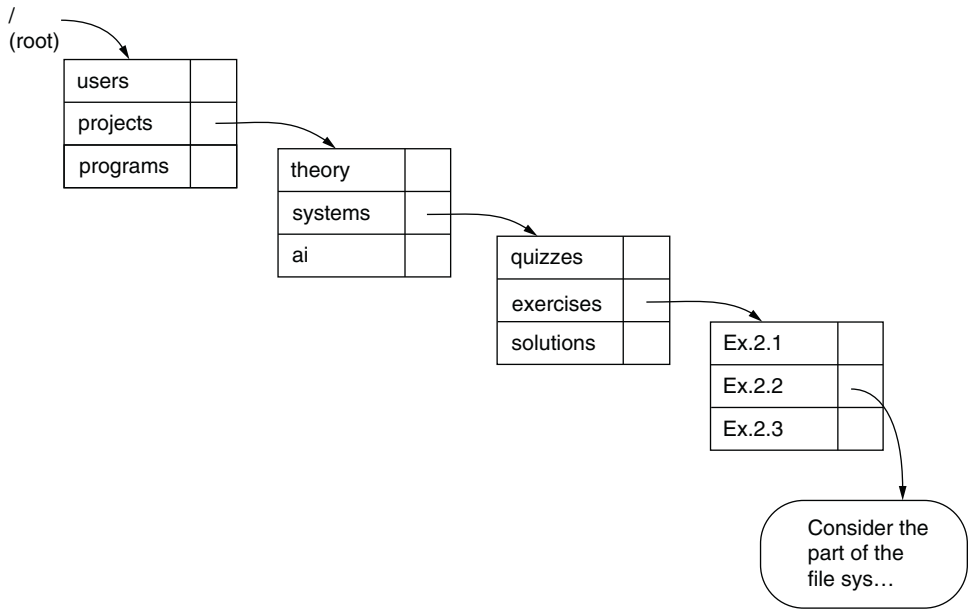
- **a.** *Follow me.* Bob is going to a party at Mary's home for the evening, so he forwards his telephone to Mary. Ann is baby-sitting for Bob, so she forwards her telephone to Bob. Jim calls Ann's number, Bob's telephone rings, and Ann answers it.
- **b.** *Delegation.* Bob is going to a party at Mary's home for the evening, so he forwards his telephone to Mary. Ann is gone for the week and has forwarded her telephone to Bob so that he can take her calls. Jim calls Ann's number, Mary's telephone rings, and Mary hands the phone to Bob to take the call.

2.1a Using the terminology of the naming section of this chapter, explain these two possibilities.

2.1b What might go wrong if Bob has already forwarded his telephone to Mary before Ann forwards her telephone to him?

2.1c The telephone company usually provides *Delegation* rather than *Follow me*. Why?

2.2 Consider the part of the file system naming hierarchy illustrated in the following:



You have been handed the following path name:

`/projects/systems/exercises/Ex.2.2`

and you are about to resolve the third component of that path name, the name `exercises`.

2.2a In the path name and in the figure, identify the context that you should use for that resolution and the context reference that allows locating that context.

2.2b Which of the terms *default*, *explicit*, *built-in*, *per-object*, and *per-name* apply to this context reference?

1995-2-1a

2.3 One way to speed up the resolving of names is to implement a cache that remembers recently looked-up {name, object} pairs.

2.3a What problems do synonyms pose for cache designers, as compared with caches that don't support synonyms?

1994-2-3

2.3b Propose a way of solving the problems if every object has a unique ID.

1994-2-3a

2.4 Louis Reasoner has become concerned about the efficiency of the search rule implementation in the Eunuchs system (an emasculated version of the UNIX system). He proposes to add a *referenced object table* (ROT), which the system

will maintain for each session of each user, set to be empty when the user logs in. Whenever the system resolves a name through of use a search path, it makes an entry in the ROT consisting of the name and the path name of that object. The "already referenced" search rule simply searches the ROT to determine if the name in question appears there. If it finds a match, then the resolver will use the associated path name from the ROT. Louis proposes to always use the "already referenced" rule first, followed by the traditional search path mechanism. He claims that the user will detect no difference, except for faster name resolution. Is Louis right?

1985-2-2

2.5 The last line of Figure 2.4 names three Web browsers as examples of interpreters. Explain how a Web browser is an interpreter by identifying its instruction reference, its repertoire, and its environment reference.

2009-0-1

Additional exercises relating to Chapter 2 can be found in the problem sets beginning on page 425.