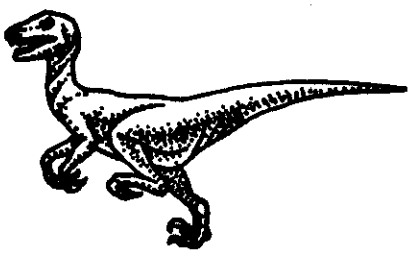
CHAPTER

I/O Systems



The two main jobs of a computer are I/O and processing. In many cases, the main job is I/O, and the processing is merely incidental. For instance, when we browse a web page or edit a file, our immediate interest is to read or enter some information, not to compute an answer.

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. Although related topics appear in other chapters, here we bring together the pieces to paint a complete picture of I/O. First, we describe the basics of I/O hardware, because the nature of the hardware interface places requirements on the internal facilities of the operating system. Next, we discuss the I/O services provided by the operating system and the embodiment of these services in the application I/O interface. Then, we explain how the operating system bridges the gap between the hardware interface and the application interface. We also discuss the UNIX System V STREAMS mechanism, which enables an application to assemble pipelines of driver code dynamically. Finally, we discuss the performance aspects of I/O and the principles of operating-system design that improve I/O performance.

13.1 Overview

The control of devices connected to the computer is a major concern of operating-system designers. Because I/O devices vary so widely in their function and speed (consider a mouse, a hard disk, and a CD-ROM jukebox), varied methods are needed to control them. These methods form the *I/O subsystem* of the kernel, which separates the rest of the kernel from the complexities of managing I/O devices.

I/O-device technology exhibits two conflicting trends. On one hand, we see increasing standardization of software and hardware interfaces. This trend helps us to incorporate improved device generations into existing computers and operating systems. On the other hand, we see an increasingly broad variety of I/O devices. Some new devices are so unlike previous devices that it is a challenge to incorporate them into our computers and operating systems. This challenge is met by a combination of hardware and software techniques. The

basic I/O hardware elements, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices. To encapsulate the details and oddities of different devices, the kernel of an operating system is structured to use device-driver modules. The **device drivers** present a uniform device-access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

13.2 I/O Hardware

Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks, tapes), transmission devices (network cards, modems), and human-interface devices (screen, keyboard, mouse). Other devices are more specialized, such as the steering of a military fighter jet or a space shuttle. In these aircraft, a human gives input to the flight computer via a joystick and foot pedals, and the computer sends output commands that cause motors to move rudders, flaps, and thrusters. Despite the incredible variety of I/O devices, though, we need only a few concepts to understand how the devices are attached and how the software can control the hardware.

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point (or **port**)—for example, a serial port. If devices use a common set of wires, the connection is called a *bus*. A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings. When device *A* has a cable that plugs into device *B*, and device *B* has a cable that plugs into device *C*, and device *C* plugs into a port on the computer, this arrangement is called a **daisy chain**. A daisy chain usually operates as a bus.

Buses are used widely in computer architecture. A typical PC bus structure appears in Figure 13.1. This figure shows a **PCI bus** (the common PC system bus) that connects the processor–memory subsystem to the fast devices and an **expansion bus** that connects relatively slow devices such as the keyboard and serial and parallel ports. In the upper-right portion of the figure, four disks are connected together on a SCSI bus plugged into a SCSI controller.

A **controller** is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a **host adapter**) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controllers. If you look at a disk drive, you will see a circuit board attached to one side. This board is the disk controller. It implements the disk side of the protocol for some kind of connection—SCSI or ATA, for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, and caching.

How can the processor give commands and data to a controller to accomplish an I/O transfer? The short answer is that the controller has one

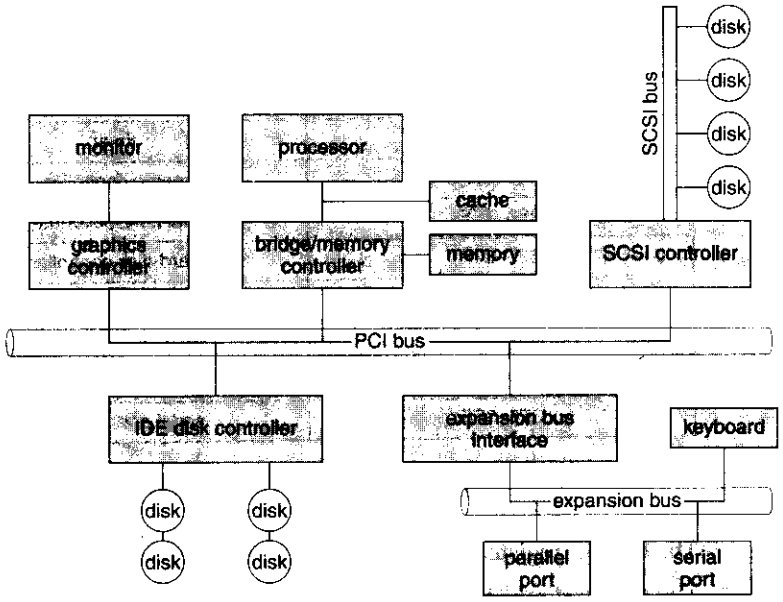


Figure 13.1 A typical PC bus structure.

or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way in which this communication can occur is through the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, the device controller can support **memory-mapped** I/O. In this case, the device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers.

Some systems use both techniques. For instance, PCs use I/O instructions to control some devices and memory-mapped I/O to control others. Figure 13.2 shows the usual I/O port addresses for PCs. The graphics controller has I/O ports for basic control operations, but the controller has a large memory-mapped region to hold screen contents. The process sends output to the screen by writing data into the memory-mapped region. The controller generates the screen image based on the contents of this memory. This technique is simple to use. Moreover, writing millions of bytes to the graphics memory is faster than issuing millions of I/O instructions. But the ease of writing to a memory-mapped I/O controller is offset by a disadvantage. Because a common type of software fault is a write through an incorrect pointer to an unintended region of memory, a memory-mapped device register is vulnerable to accidental modification. Of course, protected memory helps to reduce this risk.

An I/O port typically consists of four registers, called the (1) status, (2) control, (3) data-in, and (4) data-out registers.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Figure 13.2 Device I/O port locations on PCs (partial).

- The **data-in** register is read by the host to get input.
- The **data-out** register is written by the host to send output.
- The **status** register contains bits that can be read by the host. These bits indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
- The **control** register can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex communication, another bit enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.

The data registers are typically 1 to 4 bytes in size. Some controllers have FIFO chips that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register. A FIFO chip can hold a small burst of data until the device or host is able to receive those data.

13.2.1 Polling

The complete protocol for interaction between the host and a controller can be intricate, but the basic *handshaking* notion is simple. We explain handshaking with an example. We assume that 2 bits are used to coordinate the producer–consumer relationship between the controller and the host. The controller indicates its state through the *busy* bit in the *status* register. (Recall that to *set* a bit means to write a 1 into the bit and to *clear* a bit means to write a 0 into it.) The controller sets the *busy* bit when it is busy working and clears the *busy* bit when it is ready to accept the next command. The host signals its wishes via the *command-ready* bit in the *command* register. The host sets the *command-ready* bit when a command is available for the controller to execute. For this example,

the host writes output through a port, coordinating with the controller by handshaking as follows.

1. The host repeatedly reads the *busy* bit until that bit becomes clear.
2. The host sets the *write* bit in the *command* register and writes a byte into the *data-out* register.
3. The host sets the *command-ready* bit.
4. When the controller notices that the *command-ready* bit is set, it sets the *busy* bit.
5. The controller reads the command register and sees the `write` command. It reads the *data-out* register to get the byte and does the I/O to the device.
6. The controller clears the *command-ready* bit, clears the *error* bit in the status register to indicate that the device I/O succeeded, and clears the *busy* bit to indicate that it is finished.

This loop is repeated for each byte.

In step 1, the host is **busy-waiting** or **polling**: It is in a loop, reading the *status* register over and over until the *busy* bit becomes clear. If the controller and device are fast, this method is a reasonable one. But if the wait may be long, the host should probably switch to another task. How, then, does the host know when the controller has become idle? For some devices, the host must service the device quickly, or data will be lost. For instance, when data are streaming in on a serial port or from a keyboard, the small buffer on the controller will overflow and data will be lost if the host waits too long before returning to read the bytes.

In many computer architectures, three CPU-instruction cycles are sufficient to poll a device: *read* a device register, *logical-and* to extract a status bit, and *branch* if not zero. Clearly, the basic polling operation is efficient. But polling becomes inefficient when it is attempted repeatedly yet rarely finds a device to be ready for service, while other useful CPU processing remains undone. In such instances, it may be more efficient to arrange for the hardware controller to notify the CPU when the device becomes ready for service, rather than to require the CPU to poll repeatedly for an I/O completion. The hardware mechanism that enables a device to notify the CPU is called an **interrupt**.

13.2.2 Interrupts

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt request line, the CPU performs a state save and jumps to the **interrupt-handler** routine at a fixed address in memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a `return from interrupt` instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller *raises* an interrupt by asserting a signal on the interrupt request line, the CPU *catches* the interrupt and *dispatches* it to the interrupt handler, and the

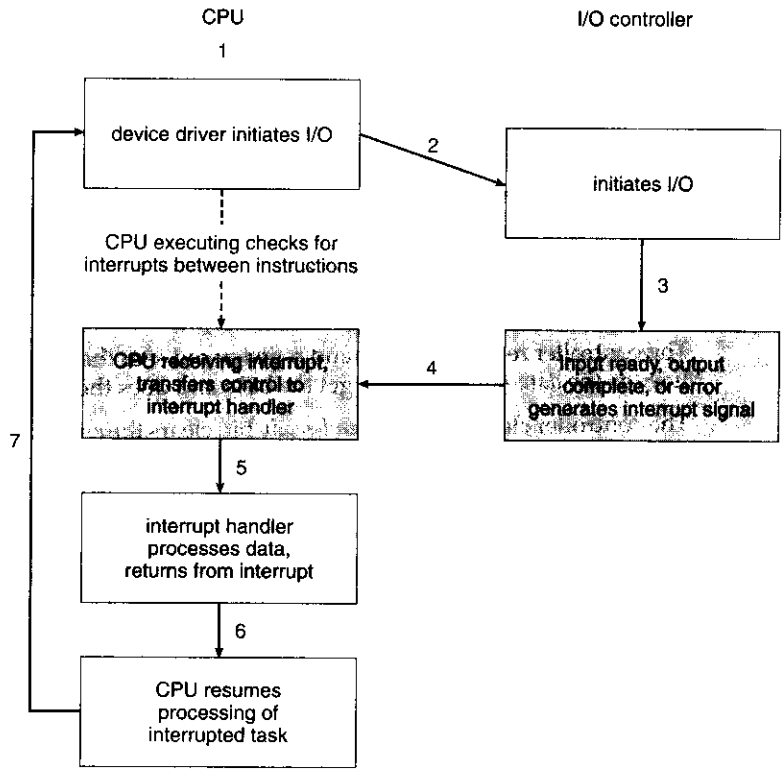


Figure 13.3 Interrupt-driven I/O cycle.

handler *clears* the interrupt by servicing the device. Figure 13.3 summarizes the interrupt-driven I/O cycle.

This basic interrupt mechanism enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. In a modern operating system, however, we need more sophisticated interrupt-handling features.

- We need the ability to defer interrupt handling during critical processing.
- We need an efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt.
- We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

In modern computer hardware, these three features are provided by the CPU and by the **interrupt-controller** hardware.

Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors.

The second interrupt line is **maskable**: It can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

The interrupt mechanism accepts an **address**—a number that selects a specific interrupt-handling routine from a small set. In most architectures, this address is an offset in a table called the **interrupt vector**. This vector contains the memory addresses of specialized interrupt handlers. The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use the technique of **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Figure 13.4 illustrates the design of the interrupt vector for the Intel Pentium processor. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

The interrupt mechanism also implements a system of **interrupt priority levels**. This mechanism enables the CPU to defer the handling of low-priority

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 13.4 Intel Pentium processor event-vector table.

interrupts without masking off all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

A modern operating system interacts with the interrupt mechanism in several ways. At boot time, the operating system probes the hardware buses to determine what devices are present and installs the corresponding interrupt handlers into the interrupt vector. During I/O, the various device controllers raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected. The interrupt mechanism is also used to handle a wide variety of **exceptions**, such as dividing by zero, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode. The events that trigger interrupts have a common property: They are occurrences that induce the CPU to execute an urgent, self-contained routine.

An operating system has other good uses for an efficient hardware and software mechanism that saves a small amount of processor state and then calls a privileged routine in the kernel. For example, many operating systems use the interrupt mechanism for virtual memory paging. A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel. This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt.

Another example is found in the implementation of system calls. Usually a program uses library calls to issue system calls. The library routines check the arguments given by the application, build a data structure to convey the arguments to the kernel, and then execute a special instruction called a **software interrupt** (or a **trap**). This instruction has an operand that identifies the desired kernel service. When a process executes the trap instruction, the interrupt hardware saves the state of the user code, switches to supervisor mode, and dispatches to the kernel routine that implements the requested service. The trap is given a relatively low interrupt priority compared with those assigned to device interrupts—executing a system call on behalf of an application is less urgent than servicing a device controller before its FIFO queue overflows and loses data.

Interrupts can also be used to manage the flow of control within the kernel. For example, consider the processing required to complete a disk read. One step is to copy data from kernel space to the user buffer. This copying is time consuming but not urgent—it should not block other high-priority interrupt handling. Another step is to start the next pending I/O for that disk drive. This step has higher priority: If the disks are to be used efficiently, we need to start the next I/O as soon as the previous one completes. Consequently, a *pair* of interrupt handlers implements the kernel code that completes a disk read. The high-priority handler records the I/O status, clears the device interrupt, starts the next pending I/O, and raises a low-priority interrupt to complete the work. Later, when the CPU is not occupied with high-priority work, the low-priority interrupt will be dispatched. The corresponding handler completes the user-level I/O by copying data from kernel buffers to the application space and then calling the scheduler to place the application on the ready queue.

A threaded kernel architecture is well suited to implement multiple interrupt priorities and to enforce the precedence of interrupt handling over

background processing in kernel and application routines. We illustrate this point with the Solaris kernel. In Solaris, interrupt handlers are executed as kernel threads. A range of high priorities is reserved for these threads. These priorities give interrupt handlers precedence over application code and kernel housekeeping and implement the priority relationships among interrupt handlers. The priorities cause the Solaris thread scheduler to preempt low-priority interrupt handlers in favor of higher-priority ones, and the threaded implementation enables multiprocessor hardware to run several interrupt handlers concurrently. We describe the interrupt architecture of UNIX and Windows XP in Appendices A and 22, respectively.

In summary, interrupts are used throughout modern operating systems to handle asynchronous events and to trap to supervisor-mode routines in the kernel. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Device controllers, hardware faults, and system calls all raise interrupts to trigger kernel routines. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system performance.

13.2.3 Direct Memory Access

For a device that does large transfers, such as a disk drive, it seems wasteful to use an expensive general-purpose processor to watch status bits and to feed data into a controller register one byte at a time—a process termed **programmed I/O (PIO)**. Many computers avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a **direct-memory-access (DMA)** controller. To initiate a DMA transfer, the host writes a DMA command block into memory. This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred. The CPU writes the address of this command block to the DMA controller, then goes on with other work. The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU. A simple DMA controller is a standard component in PCs, and **bus-mastering** I/O boards for the PC usually contain their own high-speed DMA hardware.

Handshaking between the DMA controller and the device controller is performed via a pair of wires called `DMA-request` and `DMA-acknowledge`. The device controller places a signal on the `DMA-request` wire when a word of data is available for transfer. This signal causes the DMA controller to seize the memory bus, to place the desired address on the memory-address wires, and to place a signal on the `DMA-acknowledge` wire. When the device controller receives the `DMA-acknowledge` signal, it transfers the word of data to memory and removes the `DMA-request` signal.

When the entire transfer is finished, the DMA controller interrupts the CPU. This process is depicted in Figure 13.5. When the DMA controller seizes the memory bus, the CPU is momentarily prevented from accessing main memory, although it can still access data items in its primary and secondary caches. Although this **cycle stealing** can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the total system performance. Some computer architectures use physical memory addresses for DMA, but others perform **direct virtual memory access (DVMA)**, using virtual

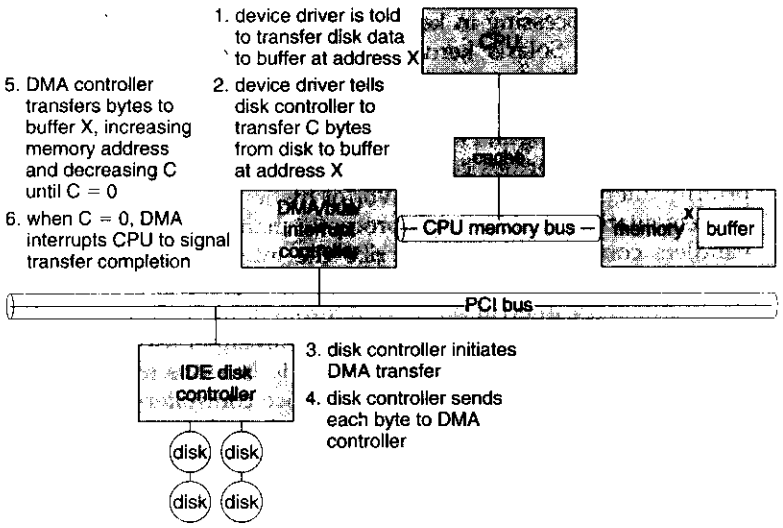


Figure 13.5 Steps in a DMA transfer.

addresses that undergo translation to physical addresses. DVMA can perform a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory.

On protected-mode kernels, the operating system generally prevents processes from issuing device commands directly. This discipline protects data from access-control violations and also protects the system from erroneous use of device controllers that could cause a system crash. Instead, the operating system exports functions that a sufficiently privileged process can use to access low-level operations on the underlying hardware. On kernels without memory protection, processes can access device controllers directly. This direct access can be used to obtain high performance, since it can avoid kernel communication, context switches, and layers of kernel software. Unfortunately, it interferes with system security and stability. The trend in general-purpose operating systems is to protect memory and devices so that the system can try to guard against erroneous or malicious applications.

13.2.4 I/O Hardware Summary

Although the hardware aspects of I/O are complex when considered at the level of detail of electronics-hardware design, the concepts that we have just described are sufficient to enable us to understand many I/O features of operating systems. Let's review the main concepts:

- A bus
- A controller
- An I/O port and its registers
- The handshaking relationship between the host and a device controller

- The execution of this handshaking in a polling loop or via interrupts
- The offloading of this work to a DMA controller for large transfers

We gave a basic example of the handshaking that takes place between a device controller and the host earlier in this section. In reality, the wide variety of available devices poses a problem for operating-system implementers. Each kind of device has its own set of capabilities, control-bit definitions, and protocols for interacting with the host—and they are all different. How can the operating system be designed so that we can attach new devices to the computer without rewriting the operating system? And when the devices vary so widely, how can the operating system give a convenient, uniform I/O interface to applications? We address those questions next.

13.3 Application I/O Interface

In this section, we discuss structuring techniques and interfaces for the operating system that enable I/O devices to be treated in a standard, uniform way. We explain, for instance, how an application can open a file on a disk without knowing what kind of disk it is and how new disks and other devices can be added to a computer without disruption of the operating system.

Like other complex software-engineering problems, the approach here involves abstraction, encapsulation, and software layering. Specifically, we can abstract away the detailed differences in I/O devices by identifying a few

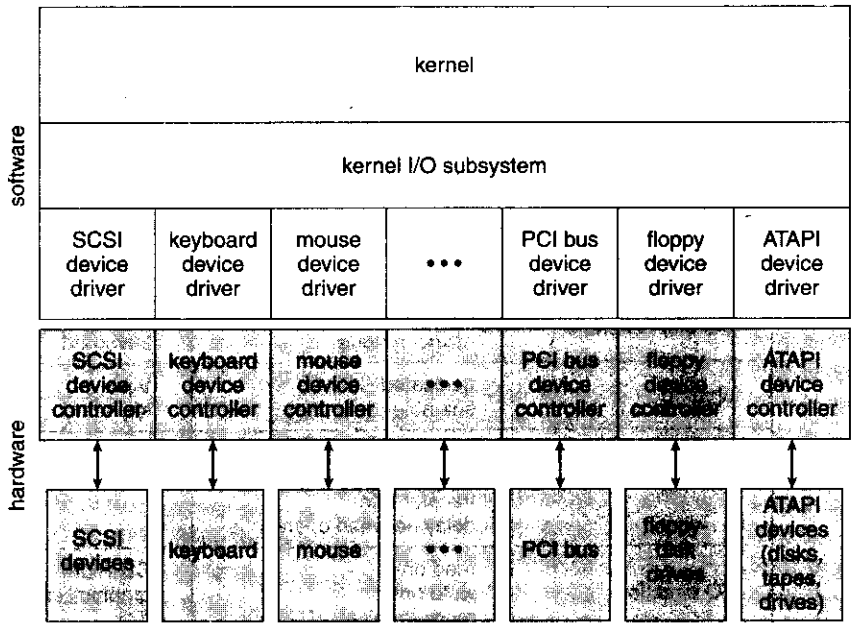


Figure 13.6 A kernel I/O structure.

general kinds. Each general kind is accessed through a standardized set of functions—an **interface**. The differences are encapsulated in kernel modules called device drivers that internally are custom-tailored to each device but that export one of the standard interfaces. Figure 13.6 illustrates how the I/O-related portions of the kernel are structured in software layers.

The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel, much as the I/O system calls encapsulate the behavior of devices in a few generic classes that hide hardware differences from applications. Making the I/O subsystem independent of the hardware simplifies the job of the operating-system developer. It also benefits the hardware manufacturers. They either design new devices to be compatible with an existing host controller interface (such as SCSI-2), or they write device drivers to interface the new hardware to popular operating systems. Thus, we can attach new peripherals to a computer without waiting for the operating-system vendor to develop support code.

Unfortunately for device-hardware manufacturers, each type of operating system has its own standards for the device-driver interface. A given device may ship with multiple device drivers—for instance, drivers for MS-DOS, Windows 95/98, Windows NT/2000, and Solaris. Devices vary on many dimensions, as illustrated in Figure 13.7.

- **Character-stream or block.** A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.
- **Sequential or random-access.** A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Figure 13.7 Characteristics of I/O devices.

- **Synchronous or asynchronous**. A synchronous device performs data transfers with predictable response times. An asynchronous device exhibits irregular or unpredictable response times.
- **Sharable or dedicated**. A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.
- **Speed of operation**. Device speeds range from a few bytes per second to a few gigabytes per second.
- **Read–write, read only, or write only**. Some devices perform both input and output, but others support only one data direction.

For the purpose of application access, many of these differences are hidden by the operating system, and the devices are grouped into a few conventional types. The resulting styles of device access have been found to be useful and broadly applicable. Although the exact system calls may differ across operating systems, the device categories are fairly standard. The major access conventions include block I/O, character-stream I/O, memory-mapped file access, and network sockets. Operating systems also provide special system calls to access a few additional devices, such as a time-of-day clock and a timer. Some operating systems provide a set of system calls for graphical display, video, and audio devices.

Most operating systems also have an **escape** (or **back door**) that transparently passes arbitrary commands from an application to a device driver. In UNIX, this system call is `ioctl()` (for "I/O" control). The `ioctl()` system call enables an application to access any functionality that can be implemented by any device driver, without the need to invent a new system call. The `ioctl()` system call has three arguments. The first is a file descriptor that connects the application to the driver by referring to a hardware device managed by that driver. The second is an integer that selects one of the commands implemented in the driver. The third is a pointer to an arbitrary data structure in memory that enables the application and driver to communicate any necessary control information or data.

13.3.1 Block and Character Devices

The **block-device** interface captures all the aspects necessary for accessing disk drives and other block-oriented devices. The device is expected to understand commands such as `read()` and `write()`; if it is a random-access device, it is also expected to have a `seek()` command to specify which block to transfer next. Applications normally access such a device through a file-system interface. We can see that `read()`, `write()`, and `seek()` capture the essential behaviors of block-storage devices, so that applications are insulated from the low-level differences among those devices.

The operating system itself, as well as special applications such as database-management systems, may prefer to access a block device as a simple linear array of blocks. This mode of access is sometimes called **raw I/O**. If the application performs its own buffering, then using a file system would cause extra, unneeded buffering. Likewise, if an application provides its own locking of file blocks or regions, then any operating-system locking services would be redundant at the least and contradictory at the worst. To avoid these conflicts,

raw-device access passes control of the device directly to the application, letting the operating system step out of the way. Unfortunately, no operating-system services are then performed on this device. A compromise that is becoming common is for the operating system to allow a mode of operation on a file that disables buffering and locking. In the UNIX world, this is called **direct I/O**.

Memory-mapped file access can be layered on top of block-device drivers. Rather than offering read and write operations, a memory-mapped interface provides access to disk storage via an array of bytes in main memory. The system call that maps a file into memory returns the virtual memory address that contains a copy of the file. The actual data transfers are performed only when needed to satisfy access to the memory image. Because the transfers are handled by the same mechanism as that used for demand-paged virtual memory access, memory-mapped I/O is efficient. Memory mapping is also convenient for programmers—access to a memory-mapped file is as simple as reading from and writing to memory. Operating systems that offer virtual memory commonly use the mapping interface for kernel services. For instance, to execute a program, the operating system maps the executable into memory and then transfers control to the entry address of the executable. The mapping interface is also commonly used for kernel access to swap space on disk.

A keyboard is an example of a device that is accessed through a **character-stream** interface. The basic system calls in this interface enable an application to `get()` or `put()` one character. On top of this interface, libraries can be built that offer line-at-a-time access, with buffering and editing services (for example, when a user types a backspace, the preceding character is removed from the input stream). This style of access is convenient for input devices such as keyboards, mice, and modems that produce data for input "spontaneously" —that is, at times that cannot necessarily be predicted by the application. This access style is also good for output devices such as printers and audio boards, which naturally fit the concept of a linear stream of bytes.

13.3.2 Network Devices

Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the `read()-write()-seek()` interface used for disks. One interface available in many operating systems, including UNIX and Windows NT, is the network **socket** interface.

Think of a wall socket for electricity: Any electrical appliance can be plugged in. By analogy, the system calls in the socket interface enable an application to create a socket, to connect a local socket to a remote address (which plugs this application into a socket created by another application), to listen for any remote application to plug into the local socket, and to send and receive packets over the connection. To support the implementation of servers, the socket interface also provides a function called `select()` that manages a set of sockets. A call to `select()` returns information about which sockets have a packet waiting to be received and which sockets have room to accept a packet to be sent. The use of `select()` eliminates the polling and busy waiting that would otherwise be necessary for network I/O. These functions encapsulate the essential behaviors of networks, greatly facilitating the creation of distributed applications that can use any underlying network hardware and protocol stack.

Many other approaches to interprocess communication and network communication have been implemented. For instance, Windows NT provides one interface to the network interface card and a second interface to the network protocols (Section C.6). In UNIX, which has a long history as a proving ground for network technology, we find half-duplex pipes, full-duplex FIFOs, full-duplex STREAMS, message queues, and sockets. Information on UNIX networking is given in Appendix A (Section A.9).

13.3.3 Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:

- Give the current time.
- Give the elapsed time.
- Set a timer to trigger operation X at time T.

These functions are used heavily by the operating system, as well as by time-sensitive applications. Unfortunately, the system calls that implement these functions are not standardized across operating systems.

The hardware to measure elapsed time and to trigger operations is called a **programmable interval timer**. It can be set to wait a certain amount of time and then generate an interrupt, and it can be set to do this once or to repeat the process to generate periodic interrupts. The scheduler uses this mechanism to generate an interrupt that will preempt a process at the end of its time slice. The disk I/O subsystem uses it to invoke the flushing of dirty cache buffers to disk periodically, and the network subsystem uses it to cancel operations that are proceeding too slowly because of network congestion or failures. The operating system may also provide an interface for user processes to use timers. The operating system can support more timer requests than the number of timer hardware channels by simulating virtual clocks. To do so, the kernel (or the timer device driver) maintains a list of interrupts wanted by its own routines and by user requests, sorted in earliest-time-first order. It sets the timer for the earliest time. When the timer interrupts, the kernel signals the requester and reloads the timer with the next earliest time.

On many computers, the interrupt rate generated by the hardware clock is between 18 and 60 ticks per second. This resolution is coarse, since a modern computer can execute hundreds of millions of instructions per second. The precision of triggers is limited by the coarse resolution of the timer, together with the overhead of maintaining virtual clocks. Furthermore, if the timer ticks are used to maintain the system time-of-day clock, the system clock can drift. In most computers, the hardware clock is constructed from a high-frequency counter. In some computers, the value of this counter can be read from a device register, in which case the counter can be considered a high-resolution clock. Although this clock does not generate interrupts, it offers accurate measurements of time intervals.

13.3.4 Blocking and Nonblocking IO

Another aspect of the system-call interface relates to the choice between blocking I/O and nonblocking I/O. When an application issues a **blocking**

system call, the execution of the application is suspended. The application is moved from the operating system's run queue to a wait queue. After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution, at which time it will receive the values returned by the system call. The physical actions performed by I/O devices are generally asynchronous—they take a varying or unpredictable amount of time. Nevertheless, most operating systems use blocking system calls for the application interface, because blocking application code is easier to understand than nonblocking application code.

Some user-level processes need **nonblocking** I/O. One example is a user interface that receives keyboard and mouse input while processing and displaying data on the screen. Another example is a video application that reads frames from a file on disk while simultaneously decompressing and displaying the output on the display.

One way an application writer can overlap execution with I/O is to write a multithreaded application. Some threads can perform blocking system calls, while others continue executing. The Solaris developers used this technique to implement a user-level library for asynchronous I/O, freeing the application writer from that task. Some operating systems provide nonblocking I/O system calls. A nonblocking call does not halt the execution of the application for an extended time. Instead, it returns quickly, with a return value that indicates how many bytes were transferred.

An alternative to a nonblocking system call is an asynchronous system call. An asynchronous call returns immediately, without waiting for the I/O to complete. The application continues to execute its code. The completion of the I/O at some future time is communicated to the application, either through the setting of some variable in the address space of the application or through the triggering of a signal or software interrupt or a call-back routine that is executed outside the linear control flow of the application. The difference between nonblocking and asynchronous system calls is that a nonblocking `read()` returns immediately with whatever data are available—the full number of bytes requested, fewer, or none at all. An asynchronous `read()` call requests a transfer that will be performed in its entirety but that will complete at some future time. These two I/O methods are shown in Figure 13.8.

A good example of nonblocking behavior is the `select()` system call for network sockets. This system call takes an argument that specifies a maximum waiting time. By setting it to 0, an application can poll for network activity without blocking. But using `select()` introduces extra overhead, because the `select()` call only checks whether I/O is possible. For a data transfer, `select()` must be followed by some kind of `read()` or `write()` command. A variation on this approach, found in Mach, is a blocking multiple-read call. It specifies desired reads for several devices in one system call and returns as soon as any one of them completes.

13.4

Kernels provide many services related to I/O. Several services—scheduling, buffering, caching, spooling, device reservation, and error handling—are provided by the kernel's I/O subsystem and build on the hardware and device-

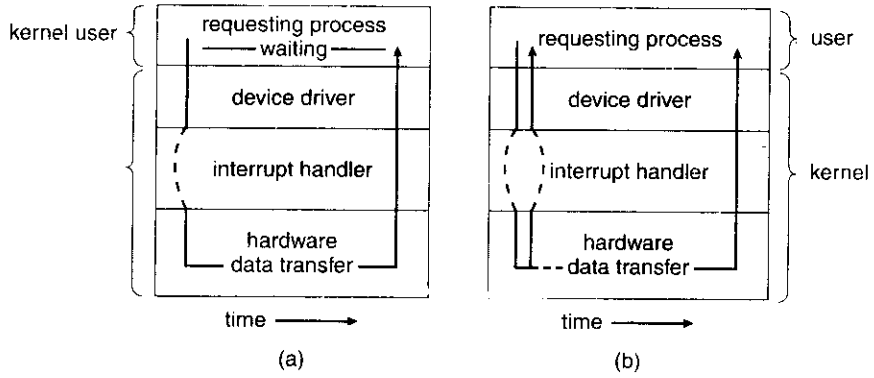


Figure 13.8 Two I/O methods: (a) synchronous and (b) asynchronous.

driver infrastructure. The I/O subsystem is also responsible for protecting itself from errant processes and malicious users.

13.4.1 I/O Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete. Here is a simple example to illustrate the opportunity. Suppose that a disk arm is near the beginning of a disk and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one near the beginning, and application 3 requests one in the middle of the disk. The operating system can reduce the distance that the disk arm travels by serving the applications in the order 2, 3, 1. Rearranging the order of service in this way is the essence of I/O scheduling.

Operating-system developers implement scheduling by maintaining a wait queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. The operating system may also try to be fair, so that no one application receives especially poor service, or it may give priority service for delay-sensitive requests. For instance, requests from the virtual memory subsystem may take priority over application requests. Several scheduling algorithms for disk I/O are detailed in Section 12.4.

When a kernel supports asynchronous I/O, it must be able to keep track of many I/O requests at the same time. For this purpose, the operating system might attach the wait queue to a **device-status table**. The kernel manages this table, which contains an entry for each I/O device, as shown in Figure 13.9. Each table entry indicates the device's type, address, and state (not functioning, idle, or busy). If the device is busy with a request, the type of request and other parameters will be stored in the table entry for that device.

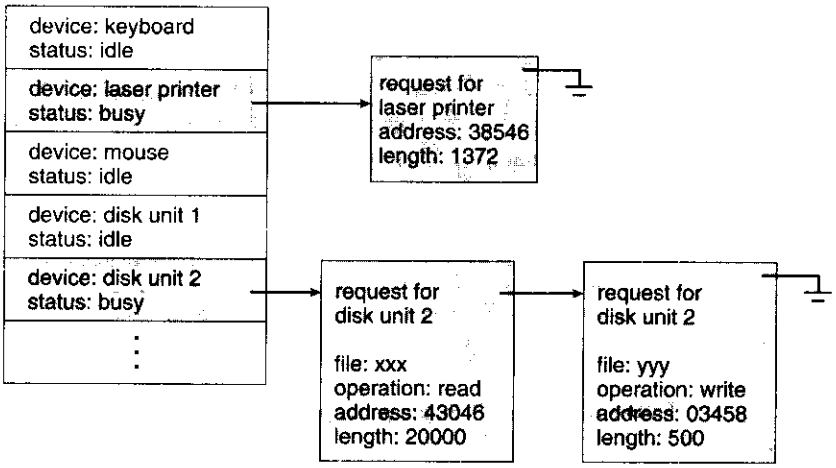


Figure 13.9 Device-status table.

One way in which the I/O subsystem improves the efficiency of the computer is by scheduling I/O operations. Another way is by using storage space in main memory or on disk via techniques called buffering, caching, and spooling.

13.4.2 Buffering

A **buffer** is a memory area that stores data while they are transferred between two devices or between a device and an application. Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. Suppose, for example, that a file is being received via modem for storage on the hard disk. The modem is about a thousand times slower than the hard disk. So a buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer of data has arrived, the buffer can be written to disk in a single operation. Since the disk write is not instantaneous and the modem still needs a place to store additional incoming data, two buffers are used. After the modem fills the first buffer, the disk write is requested. The modem then starts to fill the second buffer while the first buffer is written to disk. By the time the modem has filled the second buffer, the disk write from the first one should have completed, so the modem can switch back to the first buffer while the disk writes the second one. This **double buffering** decouples the producer of data from the consumer, thus relaxing timing requirements between them. The need for this decoupling is illustrated in Figure 13.10, which lists the enormous differences in device speeds for typical computer hardware.

A second use of buffering is to adapt between devices that have different data-transfer sizes. Such disparities are especially common in computer networking, where buffers are used widely for fragmentation and reassembly of messages. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.

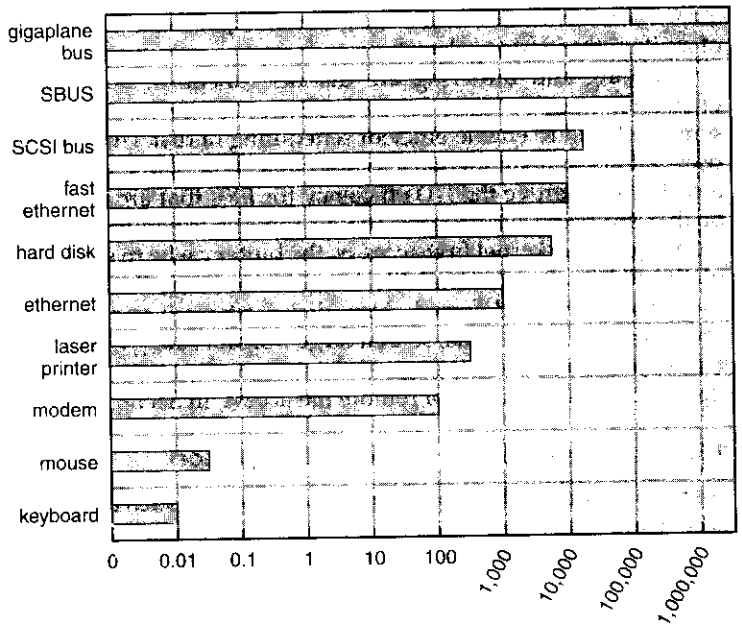


Figure 13.10 Sun Enterprise 6000 device-transfer rates (logarithmic).

A third use of buffering is to support copy semantics for application I/O. An example will clarify the meaning of "copy semantics." Suppose that an application has a buffer of data that it wishes to write to disk. It calls the `write()` system call, providing a pointer to the buffer and an integer specifying the number of bytes to write. After the system call returns, what happens if the application changes the contents of the buffer? With **copy semantics**, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application's buffer. A simple way in which the operating system can guarantee copy semantics is for the `write()` system call to copy the application data into a kernel buffer before returning control to the application. The disk write is performed from the kernel buffer, so that subsequent changes to the application buffer have no effect. Copying of data between kernel buffers and application data space is common in operating systems, despite the overhead that this operation introduces, because of the clean semantics. The same effect can be obtained more efficiently by clever use of virtual memory mapping and copy-on-write page protection.

13.4.3 Caching

A **cache** is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches. The difference between a buffer and a cache is that a buffer may hold the only existing copy of

a data item, whereas a cache, by definition, just holds a copy on faster storage of an item that resides elsewhere.

Caching and buffering are distinct functions, but sometimes a region of memory can be used for both purposes. For instance, to preserve copy semantics and to enable efficient scheduling of disk I/O, the operating system uses buffers in main memory to hold disk data. These buffers are also used as a cache, to improve the I/O efficiency for files that are shared by applications or that are being written and reread rapidly. When the kernel receives a file I/O request, the kernel first accesses the buffer cache to see whether that region of the file is already available in main memory. If so, a physical disk I/O can be avoided or deferred. Also, disk writes are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules. This strategy of delaying writes to improve I/O efficiency is discussed, in the context of remote file access, in Section 15.3.

13.4.4 Spooling and Device Reservation

A **spool** is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together. The operating system solves this problem by intercepting all output to the printer. Each application's output is spooled to a separate disk file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer. The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In others, it is handled by an in-kernel thread. In either case, the operating system provides a control interface that enables users and system administrators to display the queue, to remove unwanted jobs before those jobs print, to suspend printing while the printer is serviced, and so on.

Some devices, such as tape drives and printers, cannot usefully multiplex the I/O requests of multiple concurrent applications. Spooling is one way operating systems can coordinate concurrent output. Another way to deal with concurrent device access is to provide explicit facilities for coordination. Some operating systems (including VMS) provide support for exclusive device access by enabling a process to allocate an idle device and to deallocate that device when it is no longer needed. Other operating systems enforce a limit of one open file handle to such a device. Many operating systems provide functions that enable processes to coordinate exclusive access among themselves. For instance, Windows NT provides system calls to wait until a device object becomes available. It also has a parameter to the open() system call that declares the types of access to be permitted to other concurrent threads. On these systems, it is up to the applications to avoid deadlock.

13.4.5 Error Handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical glitch. Devices and I/O transfers can fail in many ways, either for transient reasons, as when a network becomes overloaded, or for "permanent" reasons, as when a disk controller becomes

defective. Operating systems can often compensate effectively for transient failures. For instance, a disk `read()` failure results in a `read()` retry, and a network `send()` error results in a `resend()`, if the protocol so specifies. Unfortunately, if an important component experiences a permanent failure, the operating system is unlikely to recover.

As a general rule, an I/O system call will return one bit of information about the status of the call, signifying either success or failure. In the UNIX operating system, an additional integer variable named `errno` is used to return an error code—one of about a hundred values—indicating the general nature of the failure (for example, argument out of range, bad pointer, or file not open). By contrast, some hardware can provide highly detailed error information, although many current operating systems are not designed to convey this information to the application. For instance, a failure of a SCSI device is reported by the SCSI protocol in three levels of detail: a **sense key** that identifies the general nature of the failure, such as a hardware error or an illegal request; an **additional sense code** that states the category of failure, such as a bad command parameter or a self-test failure; and an **additional sense-code qualifier** that gives even more detail, such as which command parameter was in error or which hardware subsystem failed its self-test. Further, many SCSI devices maintain internal pages of error-log information that can be requested by the host—but that seldom are.

13.4.6 I/O Protection

Errors are closely related to the issue of protection. A user process may accidentally or purposefully attempt to disrupt the normal operation of a system by attempting to issue illegal I/O instructions. We can use various mechanisms to ensure that such disruptions cannot take place in the system.

To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system. To do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf (Figure 13.11). The operating system, executing in monitor mode, checks that the request is valid and, if it is, does the I/O requested. The operating system then returns to the user.

In addition, any memory-mapped and I/O port memory locations must be protected from user access by the memory protection system. Note that a kernel cannot simply deny all user access. Most graphics games and video editing and playback software need direct access to memory-mapped graphics controller memory to speed the performance of the graphics, for example. The kernel might in this case provide a locking mechanism to allow a section of graphics memory (representing a window on screen) to be allocated to one process at a time.

13.4.7 Kernel Data Structures

The kernel needs to keep state information about the use of I/O components. It does so through a variety of in-kernel data structures, such as the open-file table structure from Section 11.1. The kernel uses many similar structures to track network connections, character-device communications, and other I/O activities.

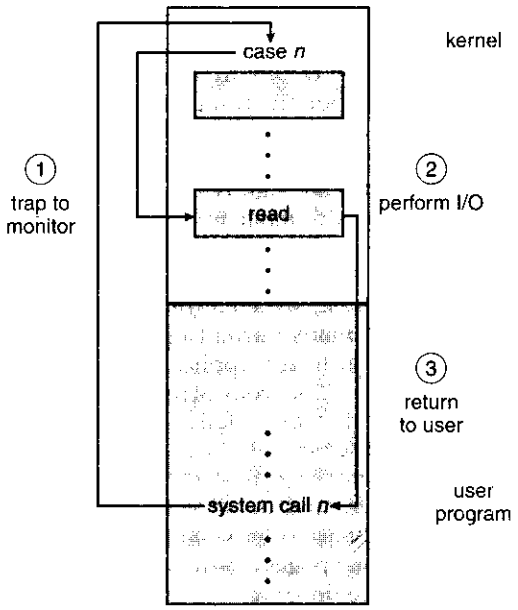


Figure 13.11 Use of a system call to perform I/O.

UNIX provides file-system access to a variety of entities, such as user files, raw devices, and the address spaces of processes. Although each of these entities supports a `read()` operation, the semantics differ. For instance, to read a user file, the kernel needs to probe the buffer cache before deciding whether to perform a disk I/O. To read a raw disk, the kernel needs to ensure that the request size is a multiple of the disk sector size and is aligned on a sector boundary. To read a process image, it is merely necessary to copy data from memory. UNIX encapsulates these differences within a uniform structure by using an object-oriented technique. The open-file record, shown in Figure 13.12, contains a dispatch table that holds pointers to the appropriate routines, depending on the type of file.

Some operating systems use object-oriented methods even more extensively. For instance, Windows NT uses a message-passing implementation for I/O. An I/O request is converted into a message that is sent through the kernel to the I/O manager and then to the device driver, each of which may change the message contents. For output, the message contains the data to be written. For input, the message contains a buffer to receive the data. The message-passing approach can add overhead, by comparison with procedural techniques that use shared data structures, but it simplifies the structure and design of the I/O system and adds flexibility.

13.4.8 Kernel I/O Subsystem Summary

In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel. The I/O subsystem supervises these procedures:

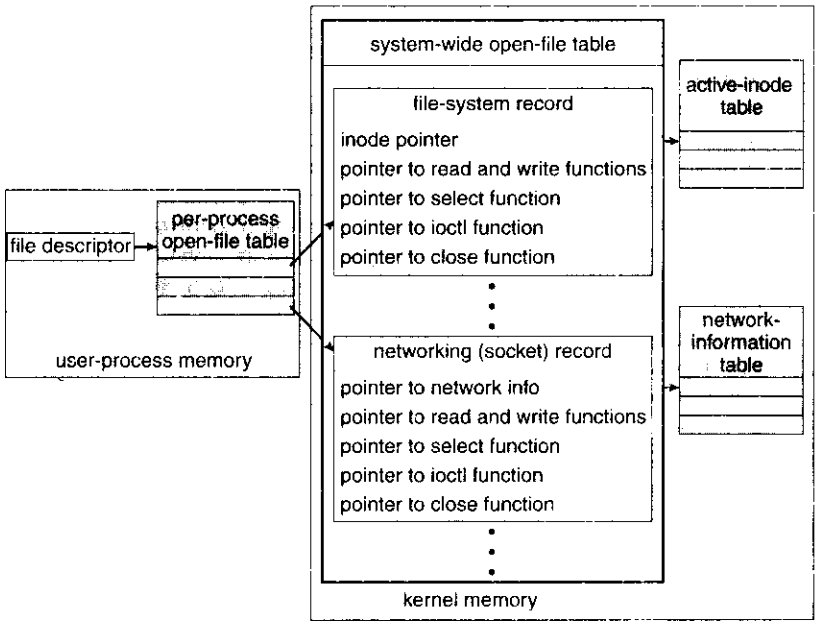


Figure 13.12 UNIX I/O kernel structure.

- Management of the name space for files and devices
- Access control to files and devices
- Operation control (for example, a modem cannot `seek()`)
- File-system space allocation
- Device allocation
- Buffering, caching, and spooling
- I/O scheduling
- Device-status monitoring, error handling, and failure recovery
- Device-driver configuration and initialization

The upper levels of the I/O subsystem access devices via the uniform interface provided by the device drivers.

13.5

Earlier, we described the handshaking between a device driver and a device controller, but we did not explain how the operating system connects an application request to a set of network wires or to a specific disk sector. Let's consider the example of reading a file from disk. The application refers to the

data by a file name. Within a disk, the file system maps from the file name through the file-system directories to obtain the space allocation of the file. For instance, in MS-DOS, the name maps to a number that indicates an entry in the file-access table, and that table entry tells which disk blocks are allocated to the file. In UNIX, the name maps to an inode number, and the corresponding inode contains the space-allocation information.

How is the connection made from the file name to the disk controller (the hardware port address or the memory-mapped controller registers)? First, we consider MS-DOS, a relatively simple operating system. The first part of an MS-DOS file name, preceding the colon, is a string that identifies a specific hardware device. For example, *c:* is the first part of every file name on the primary hard disk. The fact that *c:* represents the primary hard disk is built into the operating system; *c:* is mapped to a specific port address through a device table. Because of the colon separator, the device name space is separate from the file-system name space within each device. This separation makes it easy for the operating system to associate extra functionality with each device. For instance, it is easy to invoke spooling on any files written to the printer.

If, instead, the device name space is incorporated in the regular file-system name space, as it is in UNIX, the normal file-system name services are provided automatically. If the file system provides ownership and access control to all file names, then devices have owners and access control. Since files are stored on devices, such an interface provides access to the I/O system at two levels. Names can be used to access the devices themselves or to access the files stored on the devices.

UNIX represents device names in the regular file-system name space. Unlike an MS-DOS file name, which has a colon separator, a UNIX path name has no clear separation of the device portion. In fact, no part of the path name is the name of a device. UNIX has a **mount table** that associates prefixes of path names with specific device names. To resolve a path name, UNIX looks up the name in the mount table to find the longest matching prefix; the corresponding entry in the mount table gives the device name. This device name also has the form of a name in the file-system name space. When UNIX looks up this name in the file-system directory structures, it finds not an inode number but a <*major, minor*> device number. The major device number identifies a device driver that should be called to handle I/O to this device. The minor device number is passed to the device driver to index into a device table. The corresponding device-table entry gives the port address or the memory-mapped address of the device controller.

Modern operating systems obtain significant flexibility from the multiple stages of lookup tables in the path between a request and a physical device controller. The mechanisms that pass requests between applications and drivers are general. Thus, we can introduce new devices and drivers into a computer without recompiling the kernel. In fact, some operating systems have the ability to load device drivers on demand. At boot time, the system first probes the hardware buses to determine what devices are present; it then loads in the necessary drivers, either immediately or when first required by an I/O request.

Now we describe the typical life cycle of a blocking read request, as depicted in Figure 13.13. The figure suggests that an I/O operation requires a great many steps that together consume a tremendous number of CPU cycles.

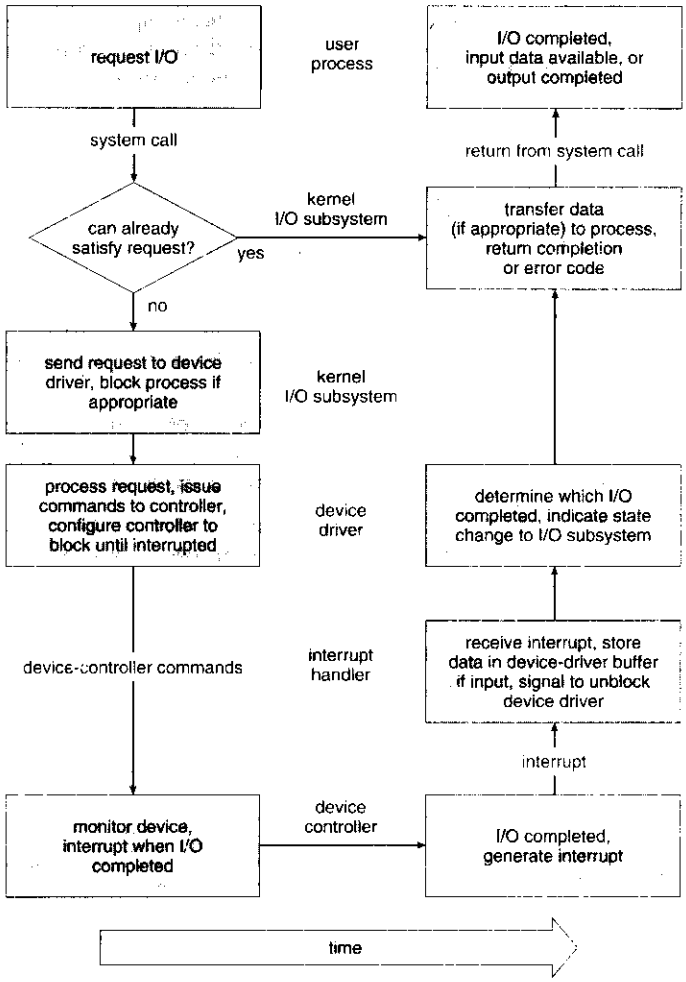


Figure 13.13 The life cycle of an I/O request.

A process issues a blocking `read()` system call to a file descriptor of a file that has been opened previously.

The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is completed.

Otherwise, a physical I/O must be performed. The process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.

The device driver allocates kernel buffer space to receive the data and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device-control registers.

The device controller operates the device hardware to perform the data transfer.

The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.

The correct interrupt handler receives the interrupt via the interrupt-vector table, stores any necessary data, signals the device driver, and returns from the interrupt.

The device driver receives the signal, determines which I/O request has completed, determines the request's status, and signals the kernel I/O subsystem that the request has been completed.

The kernel transfers data or return codes to the address space of the requesting process and moves the process from the wait queue back to the ready queue.

Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.

13.6

UNIX System V has an interesting mechanism, called **STREAMS**, that enables an application to assemble pipelines of driver code dynamically. A stream is a full-duplex connection between a device driver and a user-level process. It consists of a **stream head** that interfaces with the user process, a **driver end** that controls the device, and zero or more **stream modules** between them. The stream head, the driver end, and each module contain a pair of queues—a read queue and a write queue. Message passing is used to transfer data between queues. The STREAMS structure is shown in Figure 13.14.

Modules provide the functionality of STREAMS processing; they are *pushed* onto a stream by use of the `ioctl()` system call. For example, a process can open a serial-port device via a stream and can push on a module to handle input editing. Because messages are exchanged between queues in adjacent modules, a queue in one module may overflow an adjacent queue. To prevent this from occurring, a queue may support **flow control**. Without flow control, a queue accepts all messages and immediately sends them on to the queue in the adjacent module without buffering them. A queue supporting flow control buffers messages and does not accept messages without sufficient buffer space; this process involves exchanges of control messages between queues in adjacent modules.

A user process writes data to a device using either the `write()` or `putmsg()` system call. The `write()` system call writes raw data to the stream, whereas `putmsg()` allows the user process to specify a message. Regardless of the

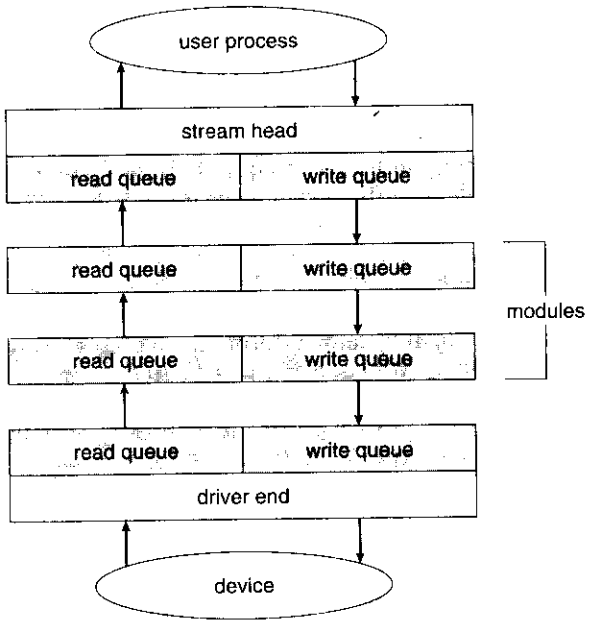


Figure 13.14 The STREAMS structure.

system call used by the user process, the stream head copies the data into a message and delivers it to the queue for the next module in line. This copying of messages continues until the message is copied to the driver end and hence the device. Similarly, the user process reads data from the stream head using either the `read()` or `getmsg()` system call. If `read()` is used, the stream head gets a message from its adjacent queue and returns ordinary data (an unstructured byte stream) to the process. If `getmsg()` is used, a message is returned to the process.

STREAMS I/O is asynchronous (or nonblocking) except when the user process communicates with the stream head. When writing to the stream, the user process will block, assuming the next queue uses flow control, until there is room to copy the message. Likewise, the user process will block when reading from the stream until data are available.

The driver end is similar to a stream head or a module in that it has a read and write queue. However, the driver end must respond to interrupts, such as one triggered when a frame is ready to be read from a network. Unlike the stream head, which may block if it is unable to copy a message to the next queue in line, the driver end must handle all incoming data. Drivers must support flow control as well. However, if a device's buffer is full, the device typically resorts to dropping incoming messages. Consider a network card whose input buffer is full. The network card must simply drop further messages until there is ample buffer space to store incoming messages.

The benefit of using STREAMS is that it provides a framework for a modular and incremental approach to writing device drivers and network protocols. Modules may be used by different streams and hence by different

devices. For example, a networking module may be used by both an Ethernet network card and a token-ring network card. Furthermore, rather than treating character-device I/O as an unstructured byte stream, STREAMS allows support for message boundaries and control information between modules. Support for STREAMS is widespread among most UNIX variants, and it is the preferred method for writing protocols and device drivers. For example, System V UNIX and Solaris implement the socket mechanism using STREAMS.

13.7

I/O is a major factor in system performance. It places heavy demands on the CPU to execute device-driver code and to schedule processes fairly and efficiently as they block and unblock. The resulting context switches stress the CPU and its hardware caches. I/O also exposes any inefficiencies in the interrupt-handling mechanisms in the kernel. In addition, I/O loads down the memory bus during data copy between controllers and physical memory and again during copies between kernel buffers and application data space. Coping gracefully with all these demands is one of the major concerns of a computer architect.

Although modern computers can handle many thousands of interrupts per second, interrupt handling is a relatively expensive task: Each interrupt causes the system to perform a state change, to execute the interrupt handler, and then to restore state. Programmed I/O can be more efficient than interrupt-driven I/O, if the number of cycles spent in busy waiting is not excessive. An I/O completion typically unblocks a process, leading to the full overhead of a context switch.

Network traffic can also cause a high context-switch rate. Consider, for instance, a remote login from one machine to another. Each character typed on the local machine must be transported to the remote machine. On the local machine, the character is typed; a keyboard interrupt is generated; and the character is passed through the interrupt handler to the device driver, to the kernel, and then to the user process. The user process issues a network I/O system call to send the character to the remote machine. The character then flows into the local kernel, through the network layers that construct a network packet, and into the network device driver. The network device driver transfers the packet to the network controller, which sends the character and generates an interrupt. The interrupt is passed back up through the kernel to cause the network I/O system call to complete.

Now, the remote system's network hardware receives the packet, and an interrupt is generated. The character is unpacked from the network protocols and is given to the appropriate network daemon. The network daemon identifies which remote login session is involved and passes the packet to the appropriate subdaemon for that session. Throughout this flow, there are context switches and state switches (Figure 13.15). Usually, the receiver echoes the character back to the sender; that approach doubles the work.

To eliminate the context switches involved in moving each character between daemons and the kernel, the Solaris developers reimplemented the **telnet** daemon using in-kernel threads. Sun estimates that this improvement increased the maximum number of network logins from a few hundred to a few thousand on a large server.

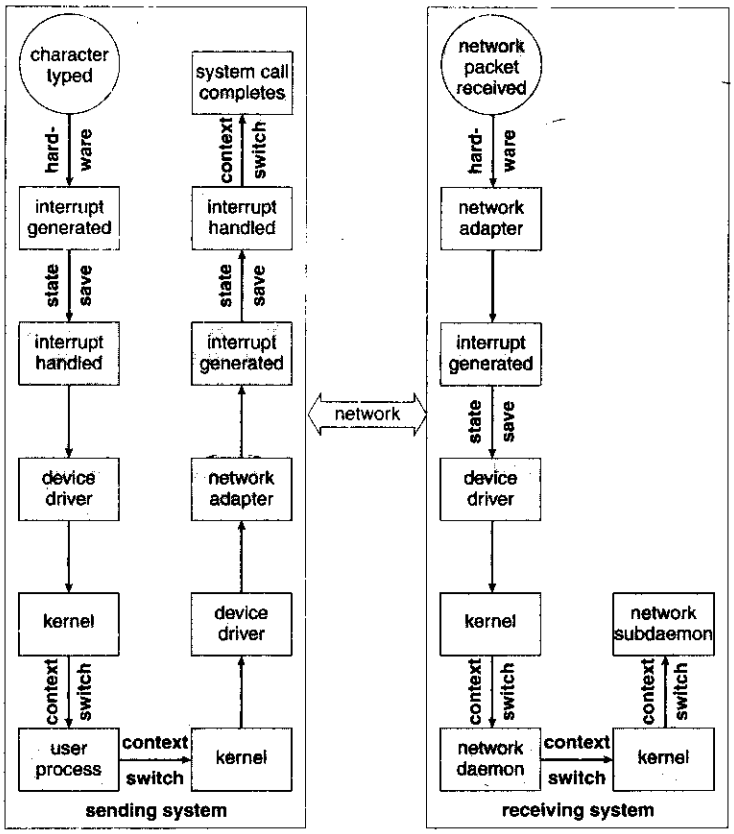


Figure 13.15 Intercomputer communications.

Other systems use separate **front-end processors** for terminal I/O to reduce the interrupt burden on the main CPU. For instance, a **terminal concentrator** can multiplex the traffic from hundreds of remote terminals into one port on a large computer. An **I/O channel** is a dedicated, special-purpose CPU found in mainframes and in other high-end systems. The job of a channel is to offload I/O work from the main CPU. The idea is that the channels keep the data flowing smoothly, while the main CPU remains free to process the data. Like the device controllers and DMA controllers found in smaller computers, a channel can process more general and sophisticated programs, so channels can be tuned for particular workloads.

We can employ several principles to improve the efficiency of I/O:

Reduce the number of context switches.

Reduce the number of times that data must be copied in memory while passing between device and application.

Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy waiting can be minimized).

Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.

Move processing primitives into hardware, to allow their operation in device controllers to be concurrent with CPU and bus operation.

Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.

Devices vary greatly in complexity. For instance, a mouse is simple. The mouse movements and button clicks are converted into numeric values that are passed from hardware, through the mouse device driver, to the application. By contrast, the functionality provided by the Windows NT disk device driver is complex. It not only manages individual disks but also implements RAID arrays (Section 12.7). To do so, it converts an application's read or write request into a coordinated set of disk I/O operations. Moreover, it implements sophisticated error-handling and data-recovery algorithms and takes many steps to optimize disk performance.

Where should the I/O functionality be implemented—in the device hardware, in the device driver, or in application software? Sometimes we observe the progression depicted in Figure 13.16.

Initially, we implement experimental I/O algorithms at the application level, because application code is flexible and application bugs are unlikely to cause system crashes. Furthermore, by developing code at the application level, we avoid the need to reboot or reload device drivers after every change to the code. An application-level implementation can be inefficient, however, because of the overhead of context switches and because the application cannot take advantage of internal kernel data structures and

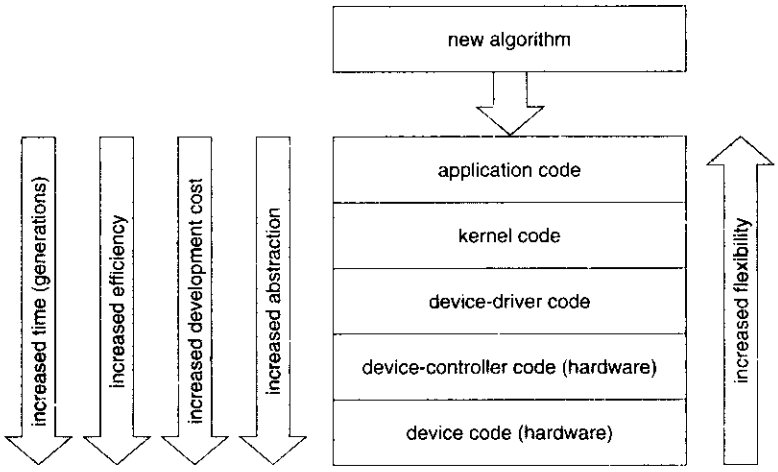


Figure 13.16 Device functionality progression.

kernel functionality (such as efficient in-kernel messaging, threading, and locking).

When an application-level algorithm has demonstrated its worth, we may reimplement it in the kernel. This can improve the performance, but the development effort is more challenging, because an operating-system kernel is a large, complex software system. Moreover, an in-kernel implementation must be thoroughly debugged to avoid data corruption and system crashes.

The highest performance may be obtained by a specialized implementation in hardware, either in the device or in the controller. The disadvantages of a hardware implementation include the difficulty and expense of making further improvements or of fixing bugs, the increased development time (months rather than days), and the decreased flexibility. For instance, a hardware RAID controller may not provide any means for the kernel to influence the order or location of individual block reads and writes, even if the kernel has special information about the workload that would enable the kernel to improve the I/O performance.

13.8

The basic hardware elements involved in I/O are buses, device controllers, and the devices themselves. The work of moving data between devices and main memory is performed by the CPU as programmed I/O or is offloaded to a DMA controller. The kernel module that controls a device is a device driver. The system-call interface provided to applications is designed to handle several basic categories of hardware, including block devices, character devices, memory-mapped files, network sockets, and programmed interval timers. The system calls usually block the process that issues them, but nonblocking and asynchronous calls are used by the kernel itself and by applications that must not sleep while waiting for an I/O operation to complete.

The kernel's I/O subsystem provides numerous services. Among these are I/O scheduling, buffering, caching, spooling, device reservation, and error handling. Another service, name translation, makes the connection between hardware devices and the symbolic file names used by applications. It involves several levels of mapping that translate from character-string names, to specific device drivers and device addresses, and then to physical addresses of I/O ports or bus controllers. This mapping may occur within the file-system name space, as it does in UNIX, or in a separate device name space, as it does in MS-DOS.

STREAMS is an implementation and methodology for making drivers reusable and easy to use. Through them, drivers can be stacked, with data passed through them sequentially and bidirectionally for processing.

I/O system calls are costly in terms of CPU consumption, because of the many layers of software between a physical device and the application. These layers imply the overheads of context switching to cross the kernel's protection boundary, of signal and interrupt handling to service the I/O devices, and of the load on the CPU and memory system to copy data between kernel buffers and application space.

13.1 When multiple interrupts from different devices appear at about the same time, a priority scheme could be used to determine the order in which the interrupts would be serviced. Discuss what issues need to be considered in assigning priorities to different interrupts.

13.2 What are the advantages and disadvantages of supporting memory-mapped I/O to device control registers?

13.3 In most multiprogrammed systems, user programs access memory through virtual addresses, while the operating system uses raw physical addresses to access memory. What are the implications of this design on the initiation of I/O operations by the user program and their execution by the operating system?

13.4 What are the various kinds of performance overheads associated with servicing an interrupt?

13.5 Typically, at the completion of a device I/O, a single interrupt is raised and appropriately handled by the host processor. In certain settings, however, the code that is to be executed at the completion of the I/O can be broken into two separate pieces, one of which executes immediately after the I/O completes and schedules a second interrupt for the remaining piece of code to be executed at a later time. What is the purpose of using this strategy in the design of interrupt handlers?

13.6 Some DMA controllers support direct virtual memory access, where the targets of I/O operations are specified as virtual addresses and a translation from virtual to physical address is performed during the DMA. How does this design complicate the design of the DMA controller? What are the advantages of providing such a functionality?

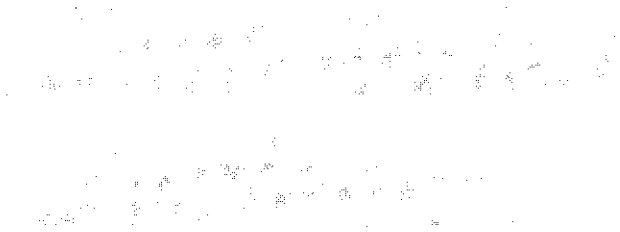
13.7 Write (in pseudocode) an implementation of virtual clocks, including the queueing and management of timer requests for the kernel and applications. Assume that the hardware provides three timer channels.

13.8 Discuss the advantages and disadvantages of guaranteeing reliable transfer of data between modules in the STREAMS abstraction.

Vahalia [1996] provides a good overview of I/O and networking in UNIX. Leffler et al. [1989] detail the I/O structures and methods employed in BSD UNIX. Milenkovic [1987] discusses the complexity of I/O methods and implementation. The use and programming of the various interprocess-communication and network protocols in UNIX are explored in Stevens [1992]. Brain [1996] documents the Windows NT application interface. The I/O implementation in the sample MINIX operating system is described in Tanenbaum and Woodhull [1997]. Custer [1994] includes detailed information on the NT message-passing implementation of I/O.

For details of hardware-level I/O handling and memory-mapping functionality, processor reference manuals (Motorola [1993] and Intel [1993]) are among the best sources. Hennessy and Patterson [2002] describe multiprocessor systems and cache-consistency issues. Tanenbaum [1990] describes hardware I/O design at a low level, and Sargent and Shoemaker [1995] provide a programmer's guide to low-level PC hardware and software. The IBM PC device I/O address map is given in IBM [1983]. The March 1994 issue of *IEEE Computer* is devoted to advanced I/O hardware and software. Rago [1993] provides a good discussion of STREAMS.

Part Six



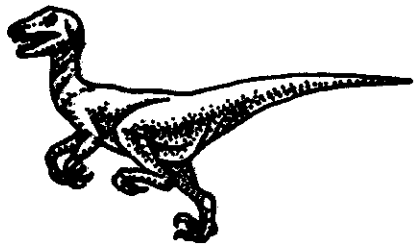
A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through communication lines such as local-area or wide-area networks. The processors in a distributed system vary in size and function. Such systems may include small handheld or real-time devices, personal computers, workstations, and large mainframe computer systems.

A distributed file system is a file-service system whose users, servers, and storage devices are dispersed among the sites of a distributed system. Accordingly, service activity has to be carried out across the network; instead of a single centralized data repository, there are multiple independent storage devices.

The benefits of a distributed system include giving users access to the resources maintained by the system and thereby speeding up computation and improving data availability and reliability. Because a system is distributed, however, it must provide mechanisms for process synchronization and communication, for dealing with the deadlock problem, and for handling failures that are not encountered in a centralized system.

Distributed Operating Systems

CHAPTER



A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication networks, such as high-speed buses or telephone lines. In this chapter, we discuss the general structure of distributed systems and the networks that interconnect them. We contrast the main differences in operating-system design between these systems and centralized systems. In Chapter 15, we go on to discuss distributed file systems. Then, in Chapter 16, we describe the methods necessary for distributed operating systems to coordinate their actions.

14.1 Motivation

A **distributed system** is a collection of loosely coupled processors interconnected by a communication network. From the point of view of a specific processor in a distributed system, the rest of the processors and their respective resources are remote, whereas its own resources are local.

The processors in a distributed system may vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems. These processors are referred to by a number of names, such as *sites*, *nodes*, *computers*, *machines*, and *hosts*, depending on the context in which they are mentioned. We mainly use *site* to indicate the location of a machine and *host* to refer to a specific system at a site. Generally, one host at one site, the *server*, has a resource that another host at another site, the *client* (or user), would like to use. A general structure of a distributed system is shown in Figure 14.1.

There are four major reasons for building distributed systems: *resource sharing*, *computation speedup*, *reliability*, and *communication*. In this section, we briefly discuss each of them.

14.1.1 Resource Sharing

If a number of different sites (with different capabilities) are connected to one another, then a user at one site may be able to use the resources available at another. For example, a user at site A may be using a laser printer located at

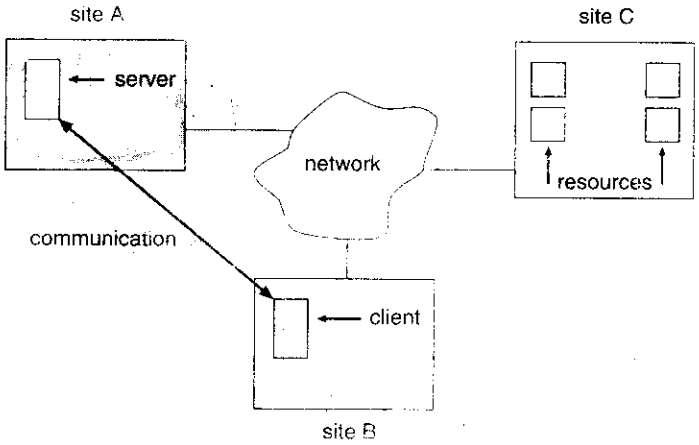


Figure 14.1 A distributed system.

site B. Meanwhile, a user at B may access a file that resides at A. In general, **resource sharing** in a distributed system provides mechanisms for sharing files at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices (such as a high-speed array processor), and performing other operations.

14.1.2 Computation Speedup

If a particular computation can be partitioned into subcomputations that can run concurrently, then a distributed system allows us to distribute the subcomputations among the various sites; the subcomputations can be run concurrently and thus provide **computation speedup**. In addition, if a particular site is currently overloaded with jobs, some of them may be moved to other, lightly loaded sites. This movement of jobs is called **load sharing**. Automated load sharing, in which the distributed operating system automatically moves jobs, is not yet common in commercial systems.

14.1.3 Reliability

If one site fails in a distributed system, the remaining sites can continue operating, giving the system better reliability. If the system is composed of multiple large autonomous installations (that is, general-purpose computers), the failure of one of them should not affect the rest. If, however, the system is composed of small machines, each of which is responsible for some crucial system function (such as terminal character I/O or the file system), then a single failure may halt the operation of the whole system. In general, with enough redundancy (in both hardware and data), the system can continue operation, even if some of its sites have failed.

The failure of a site must be detected by the system, and appropriate action may be needed to recover from the failure. The system must no longer use the services of that site. In addition, if the function of the failed site can be taken over by another site, the system must ensure that the transfer of function occurs

correctly. Finally, when the failed site recovers or is repaired, mechanisms must be available to integrate it back into the system smoothly. As we shall see in Chapters 15 and 16, these actions present difficult problems that have many possible solutions.

14.1.4 Communication

When several sites are connected to one another by a communication network, the users at different sites have the opportunity to exchange information. At a low level, **messages** are passed between systems, much as messages are passed between processes in the single-computer message system discussed in Section 3.4. Given message passing, all the higher-level functionality found in standalone systems can be expanded to encompass the distributed system. Such functions include file transfer, login, mail, and remote procedure calls (RPCs).

The advantage of a distributed system is that these functions can be carried out over great distances. Two people at geographically distant sites can collaborate on a project, for example. By transferring the files of the project, logging in to each other's remote systems to run programs, and exchanging mail to coordinate the work, users minimize the limitations inherent in long-distance work. We wrote this book by collaborating in such a manner.

The advantages of distributed systems have resulted in an industry-wide trend toward **downsizing**. Many companies are replacing their mainframes with networks of workstations or personal computers. Companies get a bigger bang for the buck (that is, better functionality for the cost), more flexibility in locating resources and expanding facilities, better user interfaces, and easier maintenance.

14.2

In this section, we describe the two general categories of network-oriented operating systems: network operating systems and distributed operating systems. Network operating systems are simpler to implement but generally more difficult for users to access and utilize than are distributed operating systems, which provide more features.

14.2.1 Network Operating Systems

A **network operating system** provides an environment in which users, who are aware of the multiplicity of machines, can access remote resources by either logging in to the appropriate remote machine or transferring data from the remote machine to their own machines.

14.2.1.1 Remote Login

An important function of a network operating system is to allow users to log in remotely. The Internet provides the **telnet** facility for this purpose. To illustrate this facility, lets suppose that a user at Westminster College wishes to compute on "cs.yale.edu," a computer that is located at Yale University. To do so, the

user must have a valid account on that machine. To log in remotely, the user issues the command

```
telnet cs.yale.edu
```

This command results in the formation of a socket connection between the local machine at Westminster College and the "cs.yale.edu" computer. After this connection has been established, the networking software creates a transparent, bidirectional link so that all characters entered by the user are sent to a process on "cs.yale.edu" and all the output from that process is sent back to the user. The process on the remote machine asks the user for a login name and a password. Once the correct information has been received, the process acts as a proxy for the user, who can compute on the remote machine just as any local user can.

14.2.1.2 Remote File Transfer

Another major function of a network operating system is to provide a mechanism for **remote file transfer** from one machine to another. In such an environment, each computer maintains its own local file system. If a user at one site (say, "cs.uvm.edu") wants to access a file located on another computer (say, "cs.yale.edu"), then the file must be copied explicitly from the computer at Yale to the computer at the University of Vermont.

The Internet provides a mechanism for such a transfer with the file transfer protocol (FTP) program. Suppose that a user on "cs.uvm.edu" wants to copy a Java program `Server.java` that resides on "cs.yale.edu." The user must first invoke the FTP program by executing

```
ftp cs.yale.edu
```

The program then asks the user for a login name and a password. Once the correct information has been received, the user must connect to the subdirectory where the file `Server.java` resides and then copy the file by executing

```
get Server.java
```

In this scheme, the file location is not transparent to the user; users must know exactly where each file is. Moreover, there is no real file sharing, because a user can only *copy* a file from one site to another. Thus, several copies of the same file may exist, resulting in a waste of space. In addition, if these copies are modified, the various copies will be inconsistent.

Notice that, in our example, the user at the University of Vermont must have login permission on "cs.yale.edu." FTP also provides a way to allow a user who does not have an account on the Yale computer to copy files remotely. This remote copying is accomplished through the "anonymous FTP" method, which works as follows. The file to be copied (that is, `Server.java`) must be placed in a special subdirectory (say, *ftp*) with the protection set to allow the public to read the file. A user who wishes to copy the file uses the `ftp` command as before. When the user is asked for the login name, the user supplies the name "anonymous" and an arbitrary password.

Once anonymous login is accomplished, care must be taken by the system to ensure that this partially authorized user does not access inappropriate

files. Generally, the user is allowed to access only those files that are in the directory tree of user "anonymous." Any files placed here are accessible to any anonymous users, subject to the usual file-protection scheme used on that machine. Anonymous users, however, cannot access files outside of this directory tree.

The FTP mechanism is implemented in a manner similar to telnet implementation. There is a daemon on the remote site that watches for connection requests to the system's FTP port. Login authentication is accomplished, and the user is allowed to execute commands remotely. Unlike the telnet daemon, which executes any command for the user, the FTP daemon responds only to a predefined set of file-related commands. These include the following:

- `get`: Transfer a file from the remote machine to the local machine.
- `put`: Transfer from the local machine to the remote machine.
- `ls` or `dir`: List files in the current directory on the remote machine.
- `cd`: Change the current directory on the remote machine.

There are also various commands to change transfer modes (for binary or ASCII files) and to determine connection status.

An important point about telnet and FTP is that they require the user to change paradigms. FTP requires the user to know a command set entirely different from the normal operating-system commands. Telnet requires a smaller shift: The user must know appropriate commands on the remote system. For instance, a user on a Windows machine who telnets to a UNIX machine must switch to UNIX commands for the duration of the telnet session. Facilities are more convenient for users if they do not require the use of a different set of commands. Distributed operating systems are designed to address this problem.

14.2.2 Distributed Operating Systems

In a distributed operating system, the users access remote resources in the same way they access local resources. Data and process migration from one site to another is under the control of the distributed operating system.

14.2.2.1 Data Migration

Suppose a user on site A wants to access data (such as a file) that reside at site B. The system can transfer the data by one of two basic methods. One approach to **data migration** is to transfer the entire file to site A. From that point on, all access to the file is local. When the user no longer needs access to the file, a copy of the file (if it has been modified) is sent back to site B. Even if only a modest change has been made to a large file, all the data must be transferred. This mechanism can be thought of as an automated FTP system. This approach was used in the Andrew file system, as we discuss in Chapter 15, but it was found to be too inefficient.

The other approach is to transfer to site A only those portions of the file that are actually *necessary* for the immediate task. If another portion is required later, another transfer will take place. When the user no longer wants to access

the file, any part of it that has been modified must be sent back to site B. (Note the similarity to demand paging.) The Sun Microsystems network file system (NFS) protocol uses this method (Chapter 15), as do newer versions of Andrew. The Microsoft SMB protocol (running on top of either TCP/IP or the Microsoft NetBEUI protocol) also allows file sharing over a network. SMB is described in Appendix C.6.1.

Clearly, if only a small part of a large file is being accessed, the latter approach is preferable. If significant portions of the file are being accessed, however, it is more efficient to copy the entire file. In both methods, data migration includes more than the mere transfer of data from one site to another. The system must also perform various data translations if the two sites involved are not directly compatible (for instance, if they use different character-code representations or represent integers with a different number or order of bits).

14.2.2.2 Computation Migration

In some circumstances, we may want to transfer the computation, rather than the data, across the system; this approach is called **computation migration**. For example, consider a job that needs to access various large files that reside at different sites, to obtain a summary of those files. It would be more efficient to access the files at the sites where they reside and return the desired results to the site that initiated the computation. Generally, if the time to transfer the data is longer than the time to execute the remote command, the remote command should be used.

Such a computation can be carried out in different ways. Suppose that process P wants to access a file at site A. Access to the file is carried out at site A and could be initiated by an RPC. An RPC uses a **datagram protocol** (UDP on the Internet) to execute a routine on a remote system (Section 3.6.2). Process P invokes a predefined procedure at site A. The procedure executes appropriately and then returns the results to P.

Alternatively, process P can send a *message* to site A. The operating system at site A then creates a new process Q whose function is to carry out the designated task. When process Q completes its execution, it sends the needed result back to P via the message system. In this scheme, process P may execute concurrently with process Q and, in fact, may have several processes running concurrently on several sites.

Both methods could be used to access several files residing at various sites. One RPC might result in the invocation of another RPC or even in the transfer of messages to another site. Similarly, process Q could, during the course of its execution, send a message to another site, which in turn would create another process. This process might either send a message back to Q or repeat the cycle.

14.2.2.3 Process Migration

A logical extension of computation migration is **process migration**. When a process is submitted for execution, it is not always executed at the site at which it is initiated. The entire process, or parts of it, may be executed at different sites. This scheme may be used for several reasons:

- **Load balancing**. The processes (or subprocesses) may be distributed across the network to even the workload.

- **Computation speedup**. If a single process can be divided into a number of subprocesses that can run concurrently on different sites, then the total process turnaround time can be reduced.
- **Hardware preference**. The process may have characteristics that make it more suitable for execution on some specialized processor (such as matrix inversion on an array processor, rather than on a microprocessor).
- **Software preference**. The process may require software that is available at only a particular site, and either the software cannot be moved, or it is less expensive to move the process.
- **Data access**. Just as in computation migration, if the data being used in the computation are numerous, it may be more efficient to have a process run remotely than to transfer all the data.

We use two complementary techniques to move processes in a computer network. In the first, the system can attempt to hide the fact that the process has migrated from the client. This scheme has the advantage that the user does not need to code her program explicitly to accomplish the migration. This method is usually employed for achieving load balancing and computation speedup among homogeneous systems, as they do not need user input to help them execute programs remotely.

The other approach is to allow (or require) the user to specify explicitly how the process should migrate. This method is usually employed when the process must be moved to satisfy a hardware or software preference.

You have probably realized that the Web has many aspects of a distributed-computing environment. Certainly it provides data migration (between a web server and a web client). It also provides computation migration. For instance, a web client could trigger a database operation on a web server. Finally, with Java, it provides a form of process migration: Java applets are sent from the server to the client, where they are executed. A network operating system provides most of these features, but a distributed operating system makes them seamless and easily accessible. The result is a powerful and easy-to-use facility—one of the reasons for the huge growth of the World Wide Web.

14.3

There are basically two types of networks: **local-area networks (LAN)** and **wide-area networks (WAN)**. The main difference between the two is the way in which they are geographically distributed. Local-area networks are composed of processors distributed over small areas (such as a single building or a number of adjacent buildings), whereas wide-area networks are composed of a number of autonomous processors distributed over a large area (such as the United States). These differences imply major variations in the speed and reliability of the communications network, and they are reflected in the distributed operating-system design.

14.3.1 Local-Area Networks

Local-area networks emerged in the early 1970s as a substitute for large mainframe computer systems. For many enterprises, it is more economical

to have a number of small computers, each with its own self-contained applications, than to have a single large system. Because each small computer is likely to need a full complement of peripheral devices (such as disks and printers), and because some form of data sharing is likely to occur in a single enterprise, it was a natural step to connect these small systems into a network.

LANs, as mentioned, are usually designed to cover a small geographical area (such as a single building or a few adjacent buildings) and are generally used in an office environment. All the sites in such systems are close to one another, so the communication links tend to have a higher speed and lower error rate than do their counterparts in wide-area networks. High-quality (expensive) cables are needed to attain this higher speed and reliability. It is also possible to use the cable exclusively for data network traffic. Over longer distances, the cost of using high-quality cable is enormous, and the exclusive use of the cable tends to be prohibitive.

The most common links in a local-area network are twisted-pair and fiber-optic cabling. The most common configurations are multiaccess bus, ring, and star networks. Communication speeds range from 1 megabit per second, for networks such as AppleTalk, infrared, and the new Bluetooth local radio network, to 1 gigabit per second for gigabit Ethernet. Ten megabits per second is most common and is the speed of **10BaseT Ethernet**. **100BaseT Ethernet** requires a higher-quality cable but runs at 100 megabits per second and is becoming common. Also growing is the use of optical-fiber–based FDDI networking. The FDDI network is token-based and runs at over 100 megabits per second.

A typical LAN may consist of a number of different computers (from mainframes to laptops or PDAs), various shared peripheral devices (such

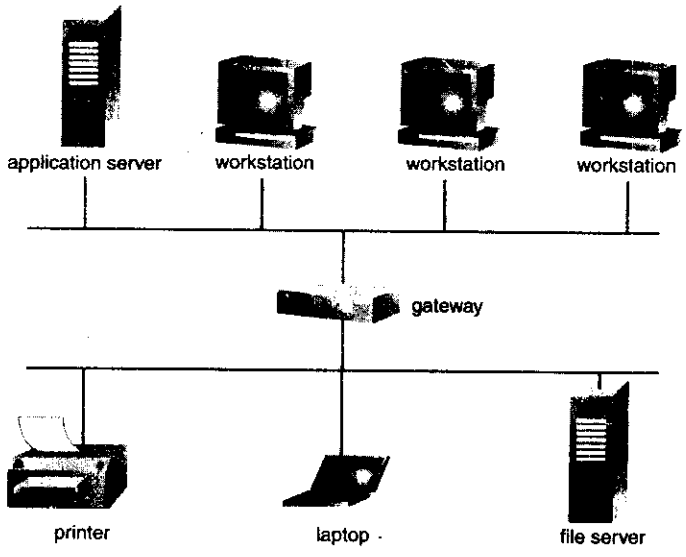


Figure 14.2 Local-area network.

as laser printers and magnetic-tape drives), and one or more gateways (specialized processors) that provide access to other networks (Figure 14.2). An Ethernet scheme is commonly used to construct LANs. An Ethernet network has no central controller, because it is a multiaccess bus, so new hosts can be added easily to the network. The Ethernet protocol is defined by the IEEE 802.3 standard.

14.3.2 Wide-Area Networks

Wide-area networks emerged in the late 1960s, mainly as an academic research project to provide efficient communication among sites, allowing hardware and software to be shared conveniently and economically by a wide community of users. The first WAN to be designed and developed was the *Arpanet*. Begun in 1968, the Arpanet has grown from a four-site experimental network to a worldwide network of networks, the Internet, comprising millions of computer systems.

Because the sites in a WAN are physically distributed over a large geographical area, the communication links are, by default, relatively slow and unreliable. Typical links are telephone lines, leased (dedicated data) lines, microwave links, and satellite channels. These communication links are controlled by special **communication processors** (Figure 14.3), which are responsible for defining the interface through which the sites communicate over the network, as well as for transferring information among the various sites.

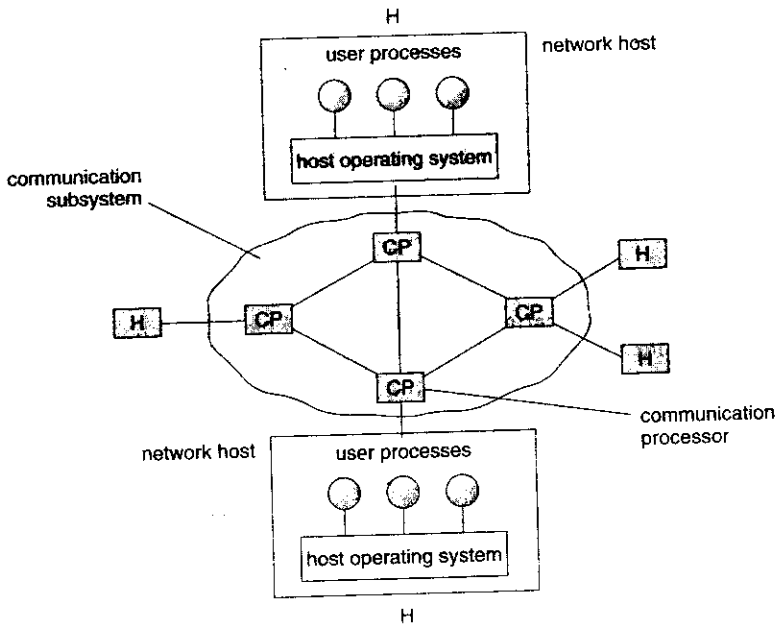


Figure 14.3 Communication processors in a wide-area network.

For example, the Internet WAN provides the ability for hosts at geographically separated sites to communicate with one another. The host computers typically differ from one another in type, speed, word length, operating system, and so on. Hosts are generally on LANs, which are, in turn, connected to the Internet via regional networks. The regional networks, such as NSFnet in the northeast United States, are interlinked with **routers** (Section 14.5.2) to form the worldwide network. Connections between networks frequently use a telephone-system service called T1, which provides a transfer rate of 1.544 megabits per second over a leased line. For sites requiring faster Internet access, T1s are collected into multiple-T1 units that work in parallel to provide more throughput. For instance, a T3 is composed of 28 T1 connections and has a transfer rate of 45 megabits per second. The routers control the path each message takes through the net. This routing may be either dynamic, to increase communication efficiency, or static, to reduce security risks or to allow communication charges to be computed.

Other WANs use standard telephone lines as their primary means of communication. **Modems** are devices that accept digital data from the computer side and convert it to the analog signals that the telephone system uses. A modem at the destination site converts the analog signal back to digital form, and the destination receives the data. The UNIX news network, UUCP, allows systems to communicate with each other at predetermined times, via modems, to exchange messages. The messages are then routed to other nearby systems and in this way either are propagated to all hosts on the network (public messages) or are transferred to their destination (private messages). WANs are generally slower than LANs; their transmission rates range from 1,200 bits per second to over 1 megabit per second. UUCP has been superseded by PPP, the point-to-point protocol. PPP functions over modem connections, allowing home computers to be fully connected to the Internet.

14.4 Network Topology

The sites in a distributed system can be connected physically in a variety of ways. Each configuration has advantages and disadvantages. We can compare the configurations by using the following criteria:

- **Installation cost.** The cost of physically linking the sites in the system
- **Communication cost.** The cost in time and money to send a message from site A to site B
- **Availability.** The extent to which data can be accessed despite the failure of some links or sites

The various topologies are depicted in Figure 14.4 as graphs whose nodes correspond to sites. An edge from node A to node B corresponds to a direct communication link between the two sites. In a fully connected network, each site is directly connected to every other site. However, the number of links grows as the square of the number of sites, resulting in a huge installation cost. Therefore, fully connected networks are impractical in any large system.

In a **partially connected network**, direct links exist between some—but not all—pairs of sites. Hence, the installation cost of such a configuration is lower than that of the fully connected network. However, if two sites A and B are not directly connected, messages from one to the other must be **routed** through a sequence of communication links. This requirement results in a higher communication cost.

If a communication link fails, messages that would have been transmitted across the link must be rerouted. In some cases, another route through the network may be found, so that the messages are able to reach their destination. In other cases, a failure may mean that no connection exists between some pairs of sites. When a system is split into two (or more) subsystems that lack any connection between them, it is partitioned. Under this definition, a subsystem (or partition) may consist of a single node.

The various partially connected network types include tree-structured networks, ring networks, and star networks, as shown in Figure 14.4. They

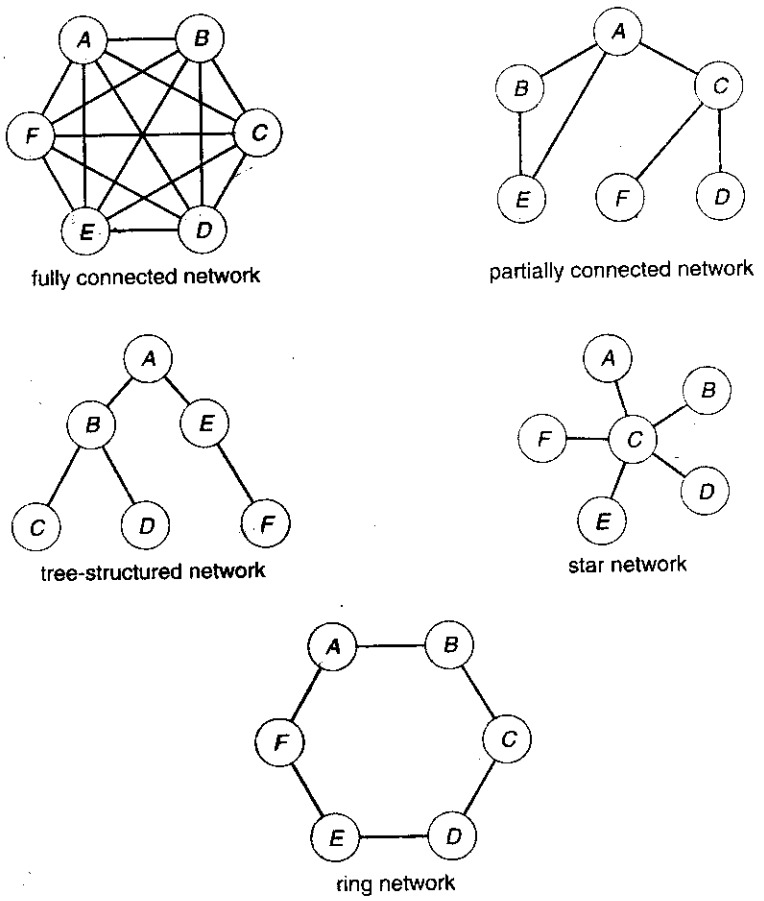


Figure 14.4 Network topology.

have different failure characteristics and installation and communication costs. Installation and communication costs are relatively low for a tree-structured network. However, the failure of a single link in such a network can result in the network's becoming partitioned. In a ring network, at least two links must fail for partition to occur. Thus, the ring network has a higher degree of availability than does a tree-structured network. However, the communication cost is high, since a message may have to cross a large number of links. In a star network, the failure of a single link results in a network partition, but one of the partitions has only a single site. Such a partition can be treated as a single-site failure. The star network also has a low communication cost, since each site is at most two links away from every other site. However, if the central site fails, every site in the system becomes disconnected.

14.5 Communication Structure

Now that we have discussed the physical aspects of networking, we turn to the internal workings. The designer of a communication network must address five basic issues:

- **Naming and name resolution**. How do two processes locate each other to communicate?
- **Routing strategies**. How are messages sent through the network?
- **Packet strategies**. Are packets sent individually or as a sequence?
- **Connection strategies**. How do two processes send a sequence of messages?
- **Contention**. How do we resolve conflicting demands for the network's use, given that it is a shared resource?

In the following sections, we elaborate on each of these issues.

14.5.1 Naming and Name Resolution

The first component of network communication is the naming of the systems in the network. For a process at site A to exchange information with a process at site B, each must be able to specify the other. Within a computer system, each process has a process identifier, and messages may be addressed with the process identifier. Because networked systems share no memory, a host within the system initially has no knowledge about the processes on other hosts.

To solve this problem, processes on remote systems are generally identified by the pair <host name, identifier>, where *host name* is a name unique within the network and *identifier* may be a process identifier or other unique number within that host. A *host name* is usually an alphanumeric identifier, rather than a number, to make it easier for users to specify. For instance, site A might have hosts named *homer, marge, bart,* and *lisa. Bart* is certainly easier to remember than is *12814831100.*

Names are convenient for humans to use, but computers prefer numbers for speed and simplicity. For this reason, there must be a mechanism to

resolve the host name into a host-id that describes the destination system to the networking hardware. This resolve mechanism is similar to the name-to-address binding that occurs during program compilation, linking, loading, and execution (Chapter 8). In the case of host names, two possibilities exist. First, every host may have a data file containing the names and addresses of all the other hosts reachable on the network (similar to binding at compile time). The problem with this model is that adding or removing a host from the network requires updating the data files on all the hosts. The alternative is to distribute the information among systems on the network. The network must then use a protocol to distribute and retrieve the information. This scheme is like execution-time binding. The first method was the original method used on the Internet; as the Internet grew, however, it became untenable, so the second method, the **domain-name system** (DNS), is now in use.

DNS specifies the naming structure of the hosts, as well as name-to-address resolution. Hosts on the Internet are logically addressed with a multipart name. Names progress from the most specific to the most general part of the address, with periods separating the fields. For instance, *bob.cs.brown.edu* refers to host *bob* in the Department of Computer Science at Brown University within the domain *edu*. (Other top-level domains include *com* for commercial sites and *org* for organizations, as well as a domain for each country connected to the network, for systems specified by country rather than organization type.) Generally, the system resolves addresses by examining the host name components in reverse order. Each component has a **name server**—simply a process on a system—that accepts a name and returns the address of the name server responsible for that name. As the final step, the name server for the host in question is contacted, and a host-id is returned. For our example system *bob.cs.brown.edu*, the following steps would be taken as result of a request made by a process on system A to communicate with *bob.cs.brown.edu*:

> The kernel of system A issues a request to the name server for the *edu* domain, asking for the address of the name server for *brown.edu*. The name server for the *edu* domain must be at a known address, so that it can be queried.
>
> The *edu* name server returns the address of the host on which the *brown.edu* name server resides.
>
> The kernel on system A then queries the name server at this address and asks about *cs.brown.edu*.
>
> An address is returned; and a request to that address for *bob.cs.brown.edu* now, finally, returns an **Internet address** host-id for that host (for example, 128.148.31.100).

This protocol may seem inefficient, but local caches are usually kept at each name server to speed the process. For example, the *edu* name server would have *brown.edu* in its cache and would inform system A that it could resolve two portions of the name, returning a pointer to the *cs.brown.edu* name server. Of course, the contents of these caches must be refreshed over time in case the name server is moved or its address changes. In fact, this service is so important that many optimizations have occurred in the protocol, as well as

many safeguards. Consider what would happen if the primary *edu* name server crashed. It is possible that no *edu* hosts would be able to have their addresses resolved, making them all unreachable! The solution is to use secondary, back-up name servers that duplicate the contents of the primary servers.

Before the domain-name service was introduced, all hosts on the Internet needed to have copies of a file that contained the names and addresses of each host on the network. All changes to this file had to be registered at one site (host SRI-NIC), and periodically all hosts had to copy the updated file from SRI-NIC to be able to contact new systems or find hosts whose addresses had changed. Under the domain-name service, each name-server site is responsible for updating the host information for that domain. For instance, any host changes at Brown University are the responsibility of the name server for *brown.edu* and do not have to be reported anywhere else. DNS lookups will automatically retrieve the updated information because *brown.edu* is contacted directly. Within domains, there can be autonomous subdomains to distribute further the responsibility for host-name and host-id changes.

Java provides the necessary API to design a program that maps IP names to IP addresses. The program shown in Figure 14.5 is passed an IP name (such as "bob.cs.brown.edu") on the command line and either outputs the IP address of the host or returns a message indicating that the host name could not be resolved. An `InetAddress` is a Java class representing an IP name or address. The static method `getByName()` belonging to the `InetAddress` class is passed a string representation of an IP name, and it returns the corresponding `InetAddress`. The program then invokes the `getHostAddress()` method, which internally uses DNS to look up the IP address of the designated host.

Generally, the operating system is responsible for accepting from its processes a message destined for <host name, identifier> and for transferring that message to the appropriate host. The kernel on the destination host is then

```
/**
 * Usage: java DNSLookUp <IP name>
 * i.e. java DNSLookUp www.wiley.com
 */
public class DNSLookUp {
  public static void main(String[] args) {
    InetAddress hostAddress;

    try {
      hostAddress = InetAddress.getByName(args[0]);
      System.out.println(hostAddress.getHostAddress());
    }
    catch (UnknownHostException uhe) {
      System.err.println("Unknown host: " + args[0]);
    }
  }
}
```

Figure 14.5 Java program illustrating a DNS lookup.

responsible for transferring the message to the process named by the identifier. This exchange is by no means trivial; it is described in Section 14.5.4.

14.5.2 Routing Strategies

When a process at site A wants to communicate with a process at site B, how is the message sent? If there is only one physical path from A to B (such as in a star or tree-structured network), the message must be sent through that path. However, if there are multiple physical paths from A to B, then several routing options exist. Each site has a **routing table** indicating the alternative paths that can be used to send a message to other sites. The table may include information about the speed and cost of the various communication paths, and it may be updated as necessary, either manually or via programs that exchange routing information. The three most common routing schemes are **fixed routing**, **virtual routing**, and **dynamic routing**.

Fixed routing. A path from A to B is specified in advance and does not change unless a hardware failure disables it. Usually, the shortest path is chosen, so that communication costs are minimized.

Virtual routing. A path from A to B is fixed for the duration of one **session**. Different sessions involving messages from A to B may use different paths. A session could be as short as a file transfer or as long as a remote-login period.

Dynamic routing. The path used to send a message from site A to site B is chosen only when a message is sent. Because the decision is made dynamically, separate messages may be assigned different paths. Site A will make a decision to send the message to site C; C, in turn, will decide to send it to site D, and so on. Eventually, a site will deliver the message to B. Usually, a site sends a message to another site on whatever link is the least used at that particular time.

There are tradeoffs among these three schemes. Fixed routing cannot adapt to link failures or load changes. In other words, if a path has been established between A and B, the messages must be sent along this path, even if the path is down or is used more heavily than another possible path. We can partially remedy this problem by using virtual routing and can avoid it completely by using dynamic routing. Fixed routing and virtual routing ensure that messages from A to B will be delivered in the order in which they were sent. In dynamic routing, messages may arrive out of order. We can remedy this problem by appending a sequence number to each message.

Dynamic routing is the most complicated to set up and run; however, it is the best way to manage routing in complicated environments. UNIX provides both fixed routing for use on hosts within simple networks and dynamic routing for complicated network environments. It is also possible to mix the two. Within a site, the hosts may just need to know how to reach the system that connects the local network to other networks (such as company-wide networks or the Internet). Such a node is known as a **gateway**. Each individual host has a static route to the gateway, although the gateway itself uses dynamic routing to reach any host on the rest of the network.

A router is the entity within the computer network responsible for routing messages. A router can be a host computer with routing software or a special-purpose device. Either way, a router must have at least two network connections, or else it would have nowhere to route messages. A router decides whether any given message needs to be passed from the network on which it is received to any other network connected to the router. It makes this determination by examining the destination Internet address of the message. The router checks its tables to determine the location of the destination host, or at least of the network to which it will send the message toward the destination host. In the case of static routing, this table is changed only by manual update (a new file is loaded onto the router). With dynamic routing, a **routing protocol** is used between routers to inform them of network changes and to allow them to update their routing tables automatically. Gateways and routers typically are dedicated hardware devices that run code out of firmware.

14.5.3 Packet Strategies

Messages are generally of variable length. To simplify the system design, we commonly implement communication with fixed-length messages called **packets**, **frames**, or **datagrams**. A communication implemented in one packet can be sent to its destination in a **connectionless message**. A connectionless message can be **unreliable**, in which case the sender has no guarantee that, and cannot tell whether, the packet reached its destination. Alternatively, the packet can be **reliable**; usually, in this case, a packet is returned from the destination indicating that the packet arrived. (Of course, the return packet could be lost along the way.) If a message is too long to fit within one packet, or if the packets need to flow back and forth between the two communicators, a connection is established to allow the reliable exchange of multiple packets.

14.5.4 Connection Strategies

Once messages are able to reach their destinations, processes can institute **communications sessions** to exchange information. Pairs of processes that want to communicate over the network can be connected in a number of ways. The three most common schemes are **circuit switching**, **message switching**, and **packet switching**.

Circuit switching. If two processes want to communicate, a permanent physical link is established between them. This link is allocated for the duration of the communication session, and no other process can use that link during this period (even if the two processes are not actively communicating for a while). This scheme is similar to that used in the telephone system. Once a communication line has been opened between two parties (that is, party A calls party B), no one else can use this circuit until the communication is terminated explicitly (for example, when the parties hang up).

Message switching. If two processes want to communicate, a temporary link is established for the duration of one message transfer. Physical links are allocated dynamically among correspondents as needed and are allocated for only short periods. Each message is a block of data

with system information—such as the source, the destination, and error-correction codes (ECC)—that allows the communication network to deliver the message to the destination correctly. This scheme is similar to the post-office mailing system. Each letter is a message that contains both the destination address and source (return) address. Many messages (from different users) can be shipped over the same link.

Packet switching. One logical message may have to be divided into a number of packets. Each packet may be sent to its destination separately, and each therefore must include a source and destination address with its data. Furthermore, the various packets may take different paths through the network. The packets must be reassembled into messages as they arrive. Note that it is not harmful for data to be broken into packets, possibly routed separately, and reassembled at the destination. Breaking up an audio signal (say, a telephone communication), in contrast, could cause great confusion if it was not done carefully.

There are obvious tradeoffs among these schemes. Circuit switching requires substantial set-up time and may waste network bandwidth, but it incurs less overhead for shipping each message. Conversely, message and packet switching require less set-up time but incur more overhead per message. Also, in packet switching, each message must be divided into packets and later reassembled. Packet switching is the method most commonly used on data networks because it makes the best use of network bandwidth.

14.5.5 Contention

Depending on the network topology, a link may connect more than two sites in the computer network, and several of these sites may want to transmit information over a link simultaneously. This situation occurs mainly in a ring or multiaccess bus network. In this case, the transmitted information may become scrambled. If it does, it must be discarded; and the sites must be notified about the problem so that they can retransmit the information. If no special provisions are made, this situation may be repeated, resulting in degraded performance. Several techniques have been developed to avoid repeated collisions, including collision detection and token passing.

CSMA/CD. Before transmitting a message over a link, a site must listen to determine whether another message is currently being transmitted over that link; this technique is called **carrier sense with multiple access (CSMA)**. If the link is free, the site can start transmitting. Otherwise, it must wait (and continue to listen) until the link is free. If two or more sites begin transmitting at exactly the same time (each thinking that no other site is using the link), then they will register a **collision detection (CD)** and will stop transmitting. Each site will try again after some random time interval. The main problem with this approach is that, when the system is very busy, many collisions may occur, and thus performance may be degraded. Nevertheless, CSMA/CD has been used successfully in the Ethernet system, the most common local area network system. One strategy for limiting the number of collisions is to limit the number of hosts per Ethernet network. Adding more hosts to a congested network could result in poor network

throughput. As systems get faster, they are able to send more packets per time segment. As a result, the number of systems per Ethernet network generally is decreasing so that networking performance is kept reasonable.

Token passing. A unique message type, known as a **token**, continuously circulates in the system (usually a ring structure). A site that wants to transmit information must wait until the token arrives. It removes the token from the ring and begins to transmit its messages. When the site completes its round of message passing, it retransmits the token. This action, in turn, allows another site to receive and remove the token and to start its message transmission. If the token gets lost, the system must then detect the loss and generate a new token. It usually does that by declaring an **election** to choose a unique site where a new token will be generated. Later, in Section 16.6, we present one election algorithm. A token-passing scheme has been adopted by the IBM and HP/Apollo systems. The benefit of a token-passing network is that performance is constant. Adding new sites to a network may lengthen the waiting time for a token, but it will not cause a large performance decrease, as may happen on Ethernet. On lightly loaded networks, however, Ethernet is more efficient, because systems can send messages at any time.

14.6

When we are designing a communication network, we must deal with the inherent complexity of coordinating asynchronous operations communicating in a potentially slow and error-prone environment. In addition, the systems on the network must agree on a protocol or a set of protocols for determining host names, locating hosts on the network, establishing connections, and so on. We can simplify the design problem (and related implementation) by partitioning the problem into multiple layers. Each layer on one system communicates with the equivalent layer on other systems. Typically, each layer has its own protocols, and communication takes place between peer layers using a specific protocol. The protocols may be implemented in hardware or software. For instance, Figure 14.6 shows the logical communications between two computers, with the three lowest-level layers implemented in hardware. Following the International Standards Organization (ISO), we refer to the layers as follows:

Physical layer. The physical layer is responsible for handling both the mechanical and the electrical details of the physical transmission of a bit stream. At the physical layer, the communicating systems must agree on the electrical representation of a binary 0 and 1, so that when data are sent as a stream of electrical signals, the receiver is able to interpret the data properly as binary data. This layer is implemented in the hardware of the networking device.

Data-link layer. The data-link layer is responsible for handling *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer.

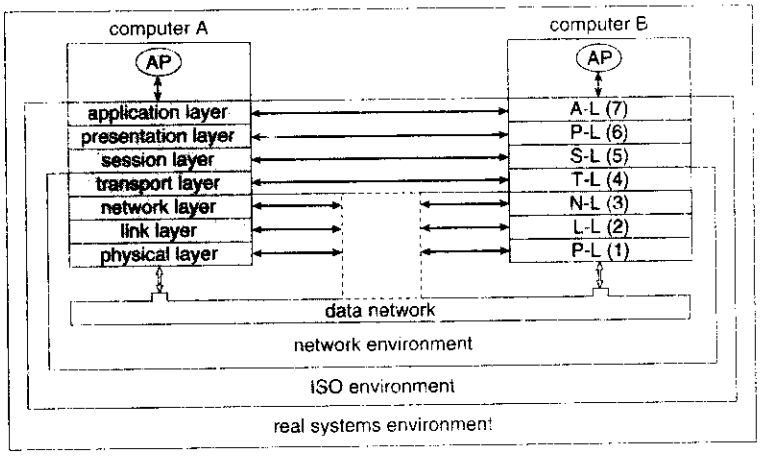


Figure 14.6 Two computers communicating via the ISO network model.

Network layer. The network layer is responsible for providing connections and for routing packets in the communication network, including handling the addresses of outgoing packets, decoding the addresses of incoming packets, and maintaining routing information for proper response to changing load levels. Routers work at this layer.

Transport layer. The transport layer is responsible for low-level access to the network and for transfer of messages between clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses.

Session layer. The session layer is responsible for implementing sessions, or process-to-process communication protocols. Typically, these protocols are the actual communications for remote logins and for file and mail transfers.

Presentation layer. The presentation layer is responsible for resolving the differences in formats among the various sites in the network, including character conversions and half duplex–full duplex modes (character echoing).

Application layer. The application layer is responsible for interacting directly with users. This layer deals with file transfer, remote-login protocols, and electronic mail, as well as with schemas for distributed databases.

Figure 14.7 summarizes the ISO **protocol stack**—a set of cooperating protocols—showing the physical flow of data. As mentioned, logically each layer of a protocol stack communicates with the equivalent layer on other systems. But physically, a message starts at or above the application layer and is passed through each lower level in turn. Each layer may modify the message and include message-header data for the equivalent layer on the receiving side. Ultimately, the message reaches the data-network layer and is transferred

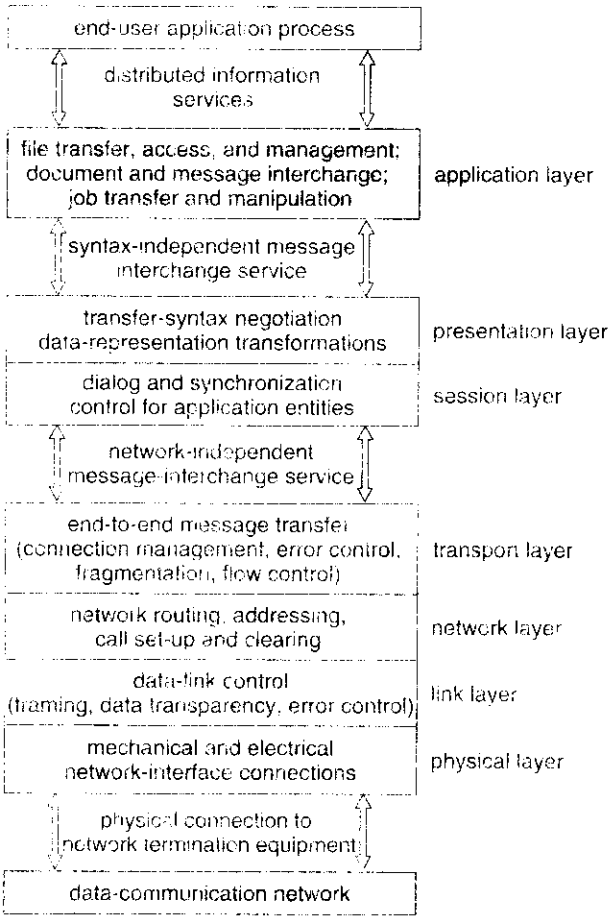


Figure 14.7 The ISO protocol stack.

as one or more packets (Figure 14.8). The data-link layer of the target system receives these data, and the message is moved up through the protocol stack; it is analyzed, modified, and stripped of headers as it progresses. It finally reaches the application layer for use by the receiving process.

The ISO model formalizes some of the earlier work done in network protocols but was developed in the late 1970s and is currently not in widespread use. Perhaps the most widely adopted protocol stack is the TCP/IP model, which has been adopted by virtually all Internet sites. The TCP/IP protocol stack has fewer layers than does the ISO model. Theoretically, because it combines several functions in each layer, it is more difficult to implement but more efficient than ISO networking. The relationship between the ISO and TCP/IP models is shown in Figure 14.9. The TCP/IP application layer identifies several protocols in widespread use in the Internet, including HTTP, FTP, Telnet, DNS, and SMTP. The transport layer identifies the unreliable, connectionless **user datagram protocol** (UDP) and the reliable, connection-oriented **transmission**

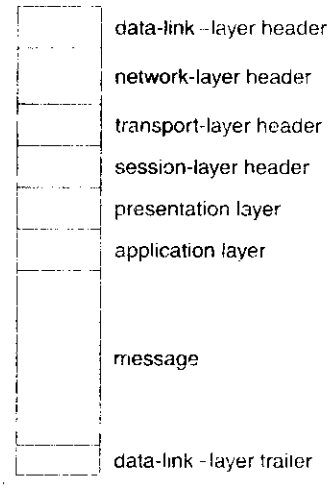


Figure 14.8 An ISO network message

control protocol (TCP). The Internet protocol (IP) is responsible for routing IP datagrams through the Internet. The TCP/IP model does not formally identify a link or physical layer, allowing TCP/IP traffic to run across any physical network. In Section 14.9, we consider the TCP/IP model running over an Ethernet network.

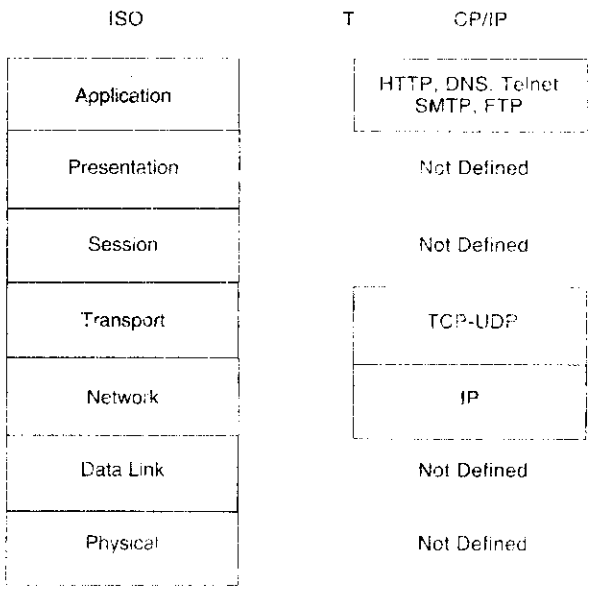


Figure 14.9 The ISO and TCP/IP protocol stacks.

14.7

A distributed system may suffer from various types of hardware failure. The failure of a link, the failure of a site, and the loss of a message are the most common types. To ensure that the system is robust, we must detect any of these failures, reconfigure the system so that computation can continue, and recover when a site or a link is repaired.

14.7.1 Failure Detection

In an environment with no shared memory, we are generally unable to differentiate among link failure, site failure, and message loss. We can usually detect only that one of these failures has occurred. Once a failure has been detected, appropriate action must be taken. What action is appropriate depends on the particular application.

To detect link and site failure, we use a **handshaking** procedure. Suppose that sites A and B have a direct physical link between them. At fixed intervals, the sites send each other an *I-am-up* message. If site A does not receive this message within a predetermined time period, it can assume that site B has failed, that the link between A and B has failed, or that the message from B has been lost. At this point, site A has two choices. It can wait for another time period to receive an *I-am-up* message from B, or it can send an *Are-you-up?* message to B.

If time goes by and site A still has not received an *I-am-up* message, or if site A has sent an *Are-you-up?* message and has not received a reply, the procedure can be repeated. Again, the only conclusion that site A can draw safely is that some type of failure has occurred.

Site A can try to differentiate between link failure and site failure by sending an *Are-you-up?* message to B by another route (if one exists). If and when B receives this message, it immediately replies positively. This positive reply tells A that B is up and that the failure is in the direct link between them. Since we do not know in advance how long it will take the message to travel from A to B and back, we must use a **time-out scheme**. At the time A sends the *Are-you-up?* message, it specifies a time interval during which it is willing to wait for the reply from B. If A receives the reply message within that time interval, then it can safely conclude that B is up. If not, however (that is, if a time-out occurs), then A may conclude only that one or more of the following situations has occurred:

Site B is down.

The direct link (if one exists) from A to B is down.

The alternative path from A to B is down.

The message has been lost.

Site A cannot, however, determine which of these events has occurred.

14.7.2 Reconfiguration

Suppose that site A has discovered, through the mechanism described in the previous section, that a failure has occurred. It must then initiate a procedure

that will allow the system to reconfigure and to continue its normal mode of operation.

If a direct link from A to B has failed, this information must be broadcast to every site in the system, so that the various routing tables can be updated accordingly.

If the system believes that a site has failed (because that site can be reached no longer), then all sites in the system must be so notified, so that they will no longer attempt to use the services of the failed site. The failure of a site that serves as a central coordinator for some activity (such as deadlock detection) requires the election of a new coordinator. Similarly, if the failed site is part of a logical ring, then a new logical ring must be constructed. Note that, if the site has not failed (that is, if it is up but cannot be reached), then we may have the undesirable situation where two sites serve as the coordinator. When the network is partitioned, the two coordinators (each for its own partition) may initiate conflicting actions. For example, if the coordinators are responsible for implementing mutual exclusion, we may have a situation where two processes are executing simultaneously in their critical sections.

14.7.3 Recovery from Failure

When a failed link or site is repaired, it must be integrated into the system gracefully and smoothly.

Suppose that a link between A and B has failed. When it is repaired, both A and B must be notified. We can accomplish this notification by continuously repeating the handshaking procedure described in Section 14.7.1.

Suppose that site B has failed. When it recovers, it must notify all other sites that it is up again. Site B then may have to receive information from the other sites to update its local tables; for example, it may need routing-table information, a list of sites that are down, or undelivered messages and mail. If the site has not failed but simply could not be reached, then this information is still required.

14.8

Making the multiplicity of processors and storage devices **transparent** to the users has been a key challenge to many designers. Ideally, a distributed system should look to its users like a conventional, centralized system. The user interface of a transparent distributed system should not distinguish between local and remote resources. That is, users should be able to access remote resources as though these resources were local, and the distributed system should be responsible for locating the resources and for arranging for the appropriate interaction.

Another aspect of transparency is user mobility. It would be convenient to allow users to log into any machine in the system rather than forcing them to use a specific machine. A transparent distributed system facilitates

user mobility by bringing over the user's environment (for example, home directory) to wherever she logs in. Both the Andrew file system from CMU and Project Athena from MIT provide this functionality on a large scale; NFS can provide it on a smaller scale.

Another design issue involves fault tolerance. We use the term *fault tolerance* in a broad sense. Communication faults, machine failures (of type fail-stop), storage-device crashes, and decays of storage media should all be tolerated to some extent. A **fault-tolerant system** should continue to function, perhaps in a degraded form, when faced with these failures. The degradation can be in performance, in functionality, or in both. It should be proportional, however, to the failures that cause it. A system that grinds to a halt when only a few of its components fail is certainly not fault tolerant. Unfortunately, fault tolerance is difficult to implement. Most commercial systems provide only limited fault tolerance. For instance, the DEC VAX cluster allows multiple computers to share a set of disks. If a system crashes, users can still access their information from another system. Of course, if a disk fails, all systems will lose access. But in this case, RAID can ensure continued access to the data even in the event of a failure (Section 12.7).

Still another issue is **scalability**—the capability of a system to adapt to increased service load. Systems have bounded resources and can become completely saturated under increased load. For example, regarding a file system, saturation occurs either when a server's CPU runs at a high utilization rate or when disks are almost full. Scalability is a relative property, but it can be measured accurately. A scalable system reacts more gracefully to increased load than does a nonscalable one. First, its performance degrades more moderately; and second, its resources reach a saturated state later. Even perfect design cannot accommodate an ever-growing load. Adding new resources might solve the problem, but it might generate additional indirect load on other resources (for example, adding machines to a distributed system can clog the network and increase service loads). Even worse, expanding the system can call for expensive design modifications. A scalable system should have the potential to grow without these problems. In a distributed system, the ability to scale up gracefully is of special importance, since expanding the network by adding new machines or interconnecting two networks is commonplace. In short, a scalable design should withstand high service load, accommodate growth of the user community, and enable simple integration of added resources.

Fault tolerance and scalability are related to each other. A heavily loaded component can become paralyzed and behave like a faulty component. Also, shifting the load from a faulty component to that component's backup can saturate the latter. Generally, having spare resources is essential for ensuring reliability as well as for handling peak loads gracefully. An inherent advantage of a distributed system is a potential for fault tolerance and scalability because of the multiplicity of resources. However, inappropriate design can obscure this potential. Fault-tolerance and scalability considerations call for a design demonstrating distribution of control and data.

Very large-scale distributed systems, to a great extent, are still only theoretical. No magic guidelines ensure the scalability of a system. It is easier to point out why current designs are *not* scalable. We next discuss several designs that pose problems and propose possible solutions, all in the context of scalability.

One principle for designing very large-scale systems is that the service demand from any component of the system should be bounded by a constant that is independent of the number of nodes in the system. Any service mechanism whose load demand is proportional to the size of the system is destined to become clogged once the system grows beyond a certain size. Adding more resources will not alleviate such a problem. The capacity of this mechanism simply limits the growth of the system.

Central control schemes and central resources should not be used to build scalable (and fault-tolerant) systems. Examples of centralized entities are central authentication servers, central naming servers, and central file servers. Centralization is a form of functional asymmetry among machines constituting the system. The ideal alternative is a functionally symmetric configuration; that is, all the component machines have an equal role in the operation of the system, and hence each machine has some degree of autonomy. Practically, it is virtually impossible to comply with such a principle. For instance, incorporating diskless machines violates functional symmetry, since the workstations depend on a central disk. However, autonomy and symmetry are important goals to which we should aspire.

The practical approximation of symmetric and autonomous configuration is **clustering**, in which the system is partitioned into a collection of semi-autonomous clusters. A **cluster** consists of a set of machines and a dedicated cluster server. So that cross-cluster resource references are relatively infrequent, each cluster server should satisfy requests of its own machines most of the time. Of course, this scheme depends on the ability to localize resource references and to place the component units appropriately. If the cluster is well balanced —that is, if the server in charge suffices to satisfy all the cluster demands—it can be used as a modular building block to scale up the system.

Deciding on the process structure of the server is a major problem in the design of any service. Servers are supposed to operate efficiently in peak periods, when hundreds of active clients need to be served simultaneously. A single-process server is certainly not a good choice, since whenever a request necessitates disk I/O, the whole service will be blocked. Assigning a process for each client is a better choice; however, the expense of frequent context switches between the processes must be considered. A related problem occurs because all the server processes need to share information.

One of the best solutions for the server architecture is the use of lightweight processes, or threads, which we discussed in Chapter 4. We can think of a group of lightweight processes as multiple threads of control associated with some shared resources. Usually, a lightweight process is not bound to a particular client. Instead, it serves single requests of different clients. Scheduling of threads can be preemptive or nonpreemptive. If threads are allowed to run to completion (nonpreemptive), then their shared data do not need to be protected explicitly. Otherwise, some explicit locking mechanism must be used. Clearly, some form of lightweight-process scheme is essential if servers are to be scalable.

14.9

We now return to the name-resolution issue raised in Section 14.5.1 and examine its operation with respect to the TCP/IP protocol stack on the Internet.

We consider the processing needed to transfer a packet between hosts on different Ethernet networks.

In a TCP/IP network, every host has a name and an associated 32-bit Internet number (or host-id). Both of these strings must be unique; and so that the name space can be managed, they are segmented. The name is hierarchical (as explained in Section 14.5.1), describing the host name and then the organization with which the host is associated. The host-id is split into a network number and a host number. The proportion of the split varies, depending on the size of the network. Once the Internet administrators assign a network number, the site with that number is free to assign host-ids.

The sending system checks its routing tables to locate a router to send the packet on its way. The routers use the network part of the host-id to transfer the packet from its source network to the destination network. The destination system then receives the packet. The packet may be a complete message, or it may just be a component of a message, with more packets needed before the message can be reassembled and passed to the TCP/UDP layer for transmission to the destination process.

Now we know how a packet moves from its source network to its destination. Within a network, how does a packet move from sender (host or router) to receiver? Every Ethernet device has a unique byte number, called the **medium access control (MAC) address**, assigned to it for addressing. Two devices on a LAN communicate with each other only with this number. If a system needs to send data to another system, the kernel generates an **address resolution protocol (ARP)** packet containing the IP address of the destination system. This packet is **broadcast** to all other systems on that Ethernet network.

A broadcast uses a special network address (usually, the maximum address) to signal that all hosts should receive and process the packet. The broadcast is not re-sent by gateways, so only systems on the local network receive it. Only the system whose IP address matches the IP address of the ARP request responds and sends back its MAC address to the system that initiated the query. For efficiency, the host caches the IP–MAC address pair in an internal table. The cache entries are **aged**, so that an entry is eventually removed from the cache if an access to that system is not required in a given time. In this way, hosts that are removed from a network are eventually *forgotten*. For added performance, ARP entries for heavily used hosts may be hardwired in the ARP cache.

Once an Ethernet device has announced its host-id and address, communication can begin. A process may specify the name of a host with which to communicate. The kernel takes that name and determines the Internet number of the target, using a DNS lookup. The message is passed from the application layer, through the software layers, and to the hardware layer. At the hardware layer, the packet (or packets) has the Ethernet address at its start; a trailer indicates the end of the packet and contains a **checksum** for detection of packet damage (Figure 14.10). The packet is placed on the network by the Ethernet device. The data section of the packet may contain some or all of the data of the original message, but it may also contain some of the upper-level headers that compose the message. In other words, all parts of the original message must be sent from source to destination, and all headers above the 802.3 layer (data-link layer) are included as data in the Ethernet packets.

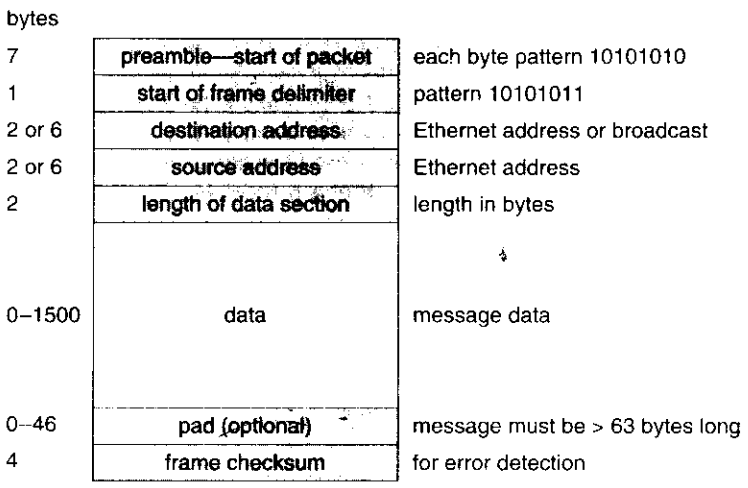


Figure 14.10 An Ethernet packet.

If the destination is on the same local network as the source, the system can look in its ARP cache, find the Ethernet address of the host, and place the packet on the wire. The destination Ethernet device then sees its address in the packet and reads in the packet, passing it up the protocol stack.

If the destination system is on a network different from that of the source, the source system finds an appropriate router on its network and sends the packet there. Routers then pass the packet along the WAN until it reaches its destination network. The router that connects the destination network checks its ARP cache, finds the Ethernet number of the destination, and sends the packet to that host. Through all of these transfers, the data-link-layer header may change as the Ethernet address of the next router in the chain is used, but the other headers of the packet remain the same until the packet is received and processed by the protocol stack and finally passed to the receiving process by the kernel.

14.10

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through various communication lines, such as high-speed buses and telephone lines. The processors in a distributed system vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems.

The processors in the system are connected through a communication network, which can be configured in a number of ways. The network may be fully or partially connected. It may be a tree, a star, a ring, or a multiaccess bus. The communication-network design must include routing and connection strategies, and it must solve the problems of contention and security.

A distributed system provides the user with access to the resources the system provides. Access to a shared resource can be provided by data migration, computation migration, or process migration.

Protocol stacks, as specified by network layering models, massage the message, adding information to it to ensure that it reaches its destination. A naming system such as DNS must be used to translate from a host name to a network address, and another protocol (such as ARP) may be needed to translate the network number to a network device address (an Ethernet address, for instance). If systems are located on separate networks, routers are needed to pass packets from source network to destination network.

A distributed system may suffer from various types of hardware failure. For a distributed system to be fault tolerant, it must detect hardware failures and reconfigure the system. When the failure is repaired, the system must be reconfigured again.

14.1 What is the difference between computation migration and process migration? Which is easier to implement, and why?

14.2 Contrast the various network topologies in terms of the following attributes:

a. Reliability

b. Available bandwidth for concurrent communications

c. Installation cost

d. Load balance in routing responsibilities

14.3 Explain why doubling the speed of the systems on an Ethernet segment may result in decreased network performance. What changes could help solve this problem?

14.4 What are the advantages of using dedicated hardware devices for routers and gateways? What are the disadvantages of using these devices compared with using general-purpose computers?

14.5 Name servers are organized in a hierarchical manner. What is the purpose of using a hierarchical organization?

14.6 Consider a network layer that senses collisions and retransmits immediately on detection of a collision. What problems could arise with this strategy? How could they be rectified?

14.7 What are the implications of using a dynamic routing strategy on application behavior? For what type of applications is it beneficial to use virtual routing instead of dynamic routing?

14.8 Run the program shown in Figure 14.5 and determine the IP addresses of the following host names:

- www.wiley.com

- www.cs.yale.edu
- www.javasoft.com
- www.westminstercollege.edu
- www.ietf.org

14.9 The original HTTP protocol used TCP/IP as the underlying network protocol. For each page, graphic, or applet, a separate TCP session was constructed, used, and torn down. Because of the overhead of building and destroying TCP/IP connections, performance problems resulted from this implementation method. Would using UDP rather than TCP be a good alternative? What other changes could you make to improve HTTP performance?

14.10 Of what use is an address-resolution protocol? Why is it better to use such a protocol than to make each host read each packet to determine that packet's destination? Does a token-passing network need such a protocol? Explain your answer.

Tanenbaum [2003], Stallings [2000a], and Kurose and Ross [2005] provided general overviews of computer networks. Williams [2001] covered computer networking from a computer-architecture viewpoint.

The Internet and its protocols were described in Comer [1999] and Comer [2000]. Coverage of TCP/IP can be found in Stevens [1994] and Stevens [1995]. UNIX network programming was described thoroughly in Stevens [1997] and Stevens [1998].

Discussions concerning distributed operating-system structures have been offered by Coulouris et al. [2001] and Tanenbaum and van Steen [2002].

Load balancing and load sharing were discussed by Harchol-Balter and Downey [1997] and Vee and Hsu [2000]. Harish and Owens [1999] described load-balancing DNS servers. Process migration was discussed by Jul et al. [1988], Douglis and Ousterhout [1991], Han and Ghosh [1998] and Milojicic et al. [2000]. Issues relating to a distributed virtual machine for distributed systems were examined in Sirer et al. [1999].