CHAPTER 11

Input/Output Systems

All computer systems must communicate with the outside world. They use a variety of devices for this purpose, depending on whether the communication partner is human or another computer. Any computer system also must provide long-term nonvolatile storage. There are several types of devices for this purpose, varying greatly in storage capacity, speed of access, and cost. The I/O system is that part of the operating system (OS) that manages communication and storage devices. It is one of the most difficult parts of the OS to design and maintain in a systematic manner. One of the main reasons for this difficulty is the range of devices that require support. In addition, the I/O system must be able to handle new devices that were not available on the market when the OS was developed or installed.

After examining basic I/O issues and models, the chapter presents a brief survey of I/O devices. Some details of device drivers are described, followed by a treatment of device management, including buffering, caching, scheduling, and error handling.

11.1 BASIC ISSUES IN DEVICE MANAGEMENT

We differentiate roughly between two types of devices: **communication devices** and **storage devices**. The former can be subdivided further into **input** devices and **output** devices.

Input devices are those that accept data generated by an external agent (e.g., a human user, a biological or chemical process, a mechanical object, or another computer) and transform data into a binary format to be stored and processed in the computer memory. The most common ones include keyboards, scanners, pointing devices (e.g., a mouse, a joystick, or a light pen), and voice analyzers. Output devices accept binary data from a computer and transform these into other formats or media to make them available to an external agent (e.g., a human user, a physical process, or another computer). These include different types of printers, plotters, visual displays, and voice synthesizers. Many of these devices provide a human-computer interface. When a computer is connected to other computers by a network, it can view the network as a communication device that can both produce input data and accept output data in various formats.

Storage devices maintain internally some data in nonvolatile form, which may be accessed and utilized by the computer. We can classify storage devices into **input/output** devices and **input-only** devices. The most common input/output storage devices are

magnetic disks and magnetic tapes; both are used for mass storage of data. Some forms of optical disks, termed CD-R or CD-RW, are also writable, but most fall into the read-only category. CD-ROMs (compact disks/read-only-memory) come with information already recorded on them by the manufacturer; the recorded data can be read with inexpensive CD-ROM drives.

All these types of communication and storage devices are considered **I/O devices**, and that part of the OS that interacts directly with these devices is the **I/O system**. A primary task of the I/O system is to make the devices usable by higher-level processes. It performs the following three basic functions:

1. presents a *logical* or *abstract view* of communication and storage devices to the users and to other high-level subsystems by hiding the details of the physical devices;
2. facilitates *efficient use* of communication and storage devices;
3. supports the convenient *sharing* of communication and storage devices.

These three functions are analogous to those formulated for file systems, which provide a high-level abstraction of secondary storage in the form of files. However, the I/O system must handle not only secondary storage, albeit at a lower level, but also all other types of communication and storage devices.

The first function addresses the fact that devices vary widely in many aspects, including the speed and granularity at which they accept or produce data, reliability, data representation, ability to be shared by multiple processes or users, and the direction of data transfer (input, output, or both). The differences are further accentuated because all devices are accessed and controlled through very low-level hardware interfaces. Each logical command issued by the user or application program must typically be decomposed into long sequences of low-level operations to trigger the actions to be performed by the I/O device and to supervise the progress of the operation by testing the device status. For example, to read a word from a disk, a sequence of instructions must be generated to move the read/write head to the track containing the desired word, await the rotational delay until the sector containing the desired word passes under the read/write head, transfer the data, and check for a number of possible error conditions. Each of these steps, in turn, consists of hundreds of instructions at the hardware device level. All such low-level details are of no interest to the processes wishing to use the I/O devices and are usually hidden from them by appropriate abstraction. Ideally, the I/O system will define only a small number of abstract device types or classes that can be accessed by a small set of read, write, and other high-level operations.

The second function above addresses performance. Most I/O devices are independent from one another and can operate concurrently with each other and with CPUs. Thus, one of the overall objectives of the I/O system is to optimize performance by *overlapping* the execution of the CPU and I/O devices as much as possible. To achieve that, the CPU must be highly responsive to the device needs so as to minimize their idle time, yet, at the same time, CPU overhead from servicing the devices cannot be too large. The task requires appropriate scheduling of I/O requests, buffering of data, and other specialized techniques.

The last point above reflects the fact that some devices can be shared concurrently by multiple processes, but other devices must be allocated exclusively to one process

at a time for a large unit of work. A typical example of the former is a disk, where multiple processes can interleave their respective read/write operations without compromising correctness. A printer, on the other hand, must be allocated to only a single process for the duration of a print job to prevent the meaningless interleaving of output from different processes. Thus, controlling access to devices is not only a matter of adequate protection (Chapters 12 and 13), but also a matter of device allocation and scheduling.

11.2 A HIERARCHICAL MODEL OF THE INPUT/OUTPUT SYSTEM

The I/O system is the layer of software that resides between the low-level hardware interfaces of individual devices and the higher-level subsystems, including the file system, the virtual memory system, and the user processes, that use the devices. Since the interfaces of most hardware devices are extremely cumbersome and also vary greatly between device classes and even individual device models, several layers of abstraction are used to bridge the gap to the high-level user and systems processes.

Figure 11-1 illustrates the relationships graphically. Each I/O device, e.g., a disk drive or a keyboard, is accessed by a special hardware device, called the **device controller**. Thus, the I/O software never communicates with any devices directly but only by issuing commands to and receiving responses from the appropriate controllers. These tasks are generally accomplished by reading and writing hardware registers provided by the controller. The set of these registers is the low-level interface between the I/O system and the controller, i.e., the **software-hardware interface**.

The I/O system can be subdivided into two layers, as indicated in the figure. The top layer consists of software that is largely independent of the specific devices being used, whereas the lower layer consists of device-specific software. Below, we briefly outline the responsibilities of each part of the I/O system.

- **Device Drivers.** Programs called *device drivers* constitute the lower layer of the I/O system. They embody specific knowledge of the devices they are to access. Given the wide variety of device types and models, device drivers must be supplied by the device manufacturers, rather than by the OS designer. Consequently, a new device driver must be installed whenever a new device is added to the system.

- **Device-Independent I/O Software.** Many functions that must be supported by the I/O system are general enough so that they apply to many types of devices and need not be modified each time a device is replaced or added. These include the tasks of data buffering, data caching, device scheduling, device naming, and others that can be provided in a device-independent manner.

User processes and other higher-level subsystems access all I/O services through a high-level *abstract* interface, which we will refer to as the **I/O system interface**.

11.2.1 The Input/Output System Interface

The I/O system typically provides only a small number of abstract device interfaces to the higher-level modules. Each such interface represents devices with similar characteristics, and provides a set of generic operations to access them. The three types shown in Figure 11-1 are representative of the possible choices.

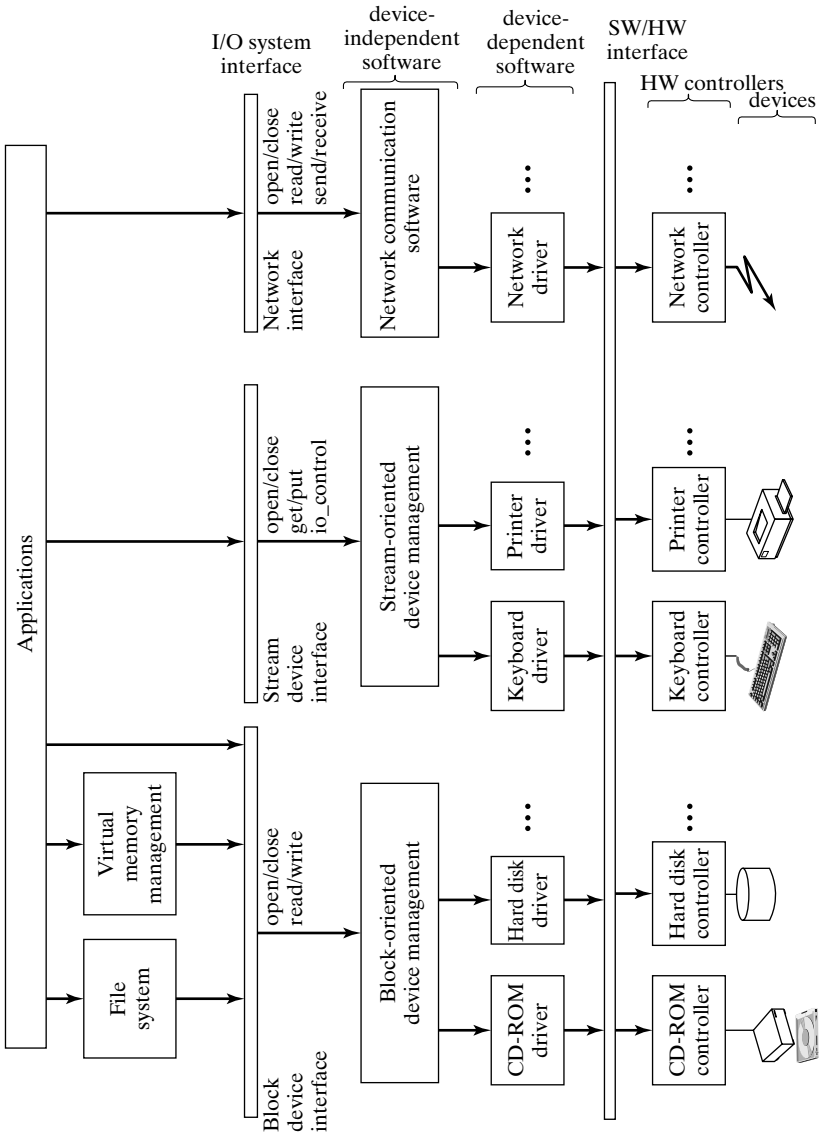

FIGURE 11-1. A hierarchical view of the I/O system.

The Block-Device Interface

The block-device interface reflects the essential characteristics of most direct-access mass-storage devices, such as magnetic or optical disks (e.g., CDs). These storage devices organize data in contiguous blocks, typically of fixed size, and support direct access to the blocks through a block number or address.

A block-device interface generally supports the following high-level operations: An ***open*** command verifies that the device is operational and prepares it for access. A ***read*** operation copies the content of the block specified by its logical block number into a region of main memory specified by an address. Similarly, a ***write*** operation overwrites the specified block with data copied from a region of main memory. A ***close*** operation releases the device if it was used exclusively or decrements the number of processes sharing it. The above high-level operations are mapped by the I/O system into lower-level operations supported by the device. For example, a read or write operation checks if the requested data is already in a disk cache in main memory. If so, it performs the access; otherwise, it initiates the necessary disk seek operation to the cylinder holding the desired block, followed by the actual data transfer operation.

Note that the block-device interface hides the fact that a disk is a two-dimensional structure consisting of cylinders and tracks within cylinders. It treats the disk as a linear sequence of blocks similar to a tape, but, unlike a tape, it makes them randomly accessible.

Some systems may permit application programs or certain system processes to access the disk directly via the block-device interface. However, since disks are typically shared, most programs are restricted to only the file system interface, which allows access to the disk only indirectly by reading and writing of files.

The **file system** uses the block-device interface to access the disk directly (Fig. 11-1). As discussed in Chapter 10, the file system translates the operations on files into the appropriate block-device commands (*read, write*) that use disk block numbers as parameters. The task of the I/O system is to translate the logical block numbers into actual disk addresses, and to initiate and supervise the transfer of the data blocks between the device and main memory.

The **virtual memory** system also uses the block-device interface. When invoked as the result of a page fault (Chapter 8), it must find the necessary page (represented as one or more data blocks on disk), and use the I/O system to read these blocks into main memory. If a page is evicted from main memory as part of the page fault, the virtual memory system must write this page back to disk by calling the appropriate *seek* and *write* commands of the I/O system.

The Stream-Device Interface

The second major interface is the **stream-device** interface, frequently also referred to as the **character-device** interface. It controls devices that produce or consume streams of characters, generally of arbitrary lengths. The individual characters must be processed in the order in which they appear in the stream and are not addressed directly. Instead, the interface supports a ***get*** and a ***put*** operation. The former returns the next character of an input stream to the caller, whereas the latter appends a new character to the output stream. This interface is representative of many communication devices, such as keyboards, pointing devices, display terminal, and printers.

In addition to the basic *get/put* operations, each device has a multitude of other functions it can perform, that vary greatly between different kinds of devices. For example, a terminal can ring the bell, toggle reverse video, or start flashing the cursor; a modem can be initialized or disconnected; and a printer can change its internal font tables and many other settings. To handle all these diverse functionalities in a uniform manner, the interface generally provides a generic ***io_control*** instruction that accepts many different parameters to express the many device-specific functions.

Finally, ***open*** and ***close*** operations are provided to reserve and release a device, since most stream-oriented devices cannot be shared but must be reserved by a process to prevent interleaving of different *read* or *write* operations.

Magnetic tapes present a special challenge. They are **block-oriented** in that each *read* or *write* operation transfers a block of data, where blocks can be of fixed or variable length. Unlike disks, however, tapes are inherently sequential in data access; their *read/write* commands do not specify the block number, which makes them **stream-oriented**. A preferred solution for tapes is to extend the stream-oriented interface so that the commands can request not only single characters one at a time (the *get/put* operations) but also read/write sequential blocks of data of arbitrary length. Other useful extensions generally also are added to the interface, for example, a ***rewind*** command to permit rereading or rewriting a tape from the beginning, or a (logical) ***seek*** command to skip over a number of blocks.

The Network Communication Interface

Increasingly, more computers are being connected to communication networks. This can be done via a **modem**, which uses telephone lines to transmit data, or specialized networks, such as an Ethernet, which is accessed via a corresponding Ethernet controller. Regardless of the connection, the network enables a computer to communicate with other computers or devices, including printers or terminals. This architecture offers great flexibility in configuring computer systems, since these devices are not dedicated to any specific computer but can be shared by many. In particular, it becomes economically feasible to provide expensive printers or other specialized devices that could not be justified for any single machine. Networking also makes it possible for users to access their computers from different locations and with different terminals. The **X terminal**, which is a sophisticated graphics terminal with its own internal processes, a keyboard, and a mouse, is a good example of a terminal that is not dedicated to any one specific computer but can communicate with different computers via the network.

To use a device that is not connected directly to a computer, the system must first establish a connection to the device. This requires that both the computer and the device name each other using network-wide identifiers, and that both express the desire to communicate with one another. A common abstraction for such a connection, supported in UNIX, Windows NT, and other systems, is the **socket**. A socket may be viewed as the endpoint of a connection from which data can be received or to which data can be sent. Thus, two processes—one on the host computer and the other on the device—that wish to communicate, each create a socket and **bind** it to each other's network address. The two commands, to create the socket and to bind it to actual addresses, are part of the high-level interface. Once a connection is established, the processes can use different communication protocols to exchange data with each other.

The two basic types of communication protocols are **connection-less** and **connection-based** protocols. In the first case, a process can simply ***send*** a message to the other process by writing it to the open socket. The other process can retrieve it from its socket using a corresponding ***receive*** command. A typical example of a connection-less protocol is UDP/IP (Universal Datagram Protocol/Internet Protocol), developed by the U.S. Department of Defense (DoD). Here, the messages are termed **datagrams**.

In connection-based communication, the two processes must first establish a higher-level connection. This is initiated by the process wishing to send data by issuing a ***connect*** command. The receiving process completes the connection by issuing an ***accept*** command. Establishing such a one-way connection is comparable to opening a shared file between the two processes. The sending process uses a ***write*** command, similar to a sequential file *write* command; the effect is to append the data to the "shared file." The receiving process uses an analogous sequential ***read*** command; the specified number of unread bytes are returned from the "shared file." The most common and widely used representative of a connection-based protocol is TCP/IP (Transmission Control Protocol/Internet Protocol), also originally developed by the U.S. DoD.

One of the main tasks of the I/O system is to translate the high-level operations that constitute its interface into low-level operations on the hardware devices. To better understand the specific tasks that the I/O system must perform at each level, we first describe the characteristics and the operating principles of the most common devices.

11.3 INPUT/OUTPUT DEVICES

11.3.1 User Terminals

The most basic user terminal consists of a keyboard for input and a visual display monitor for output. Most present-day systems also include a mouse as a pointing and selection device. Other common alternatives for user on-line interaction include joysticks or trackballs. Note that the word 'terminal' is frequently used to refer only to the display monitor. At other times, an entire PC that is connected to a network is said to be a terminal. In this chapter, we consider a terminal to be the combination of a monitor, a keyboard, and possibly a mouse.

Monitors

Monitors are similar to TV sets in that they display information dynamically on a visual screen. Most monitors use the same technology as TV sets. The inside of the screen is coated with a special chemical, which glows for a short period of time when activated by a stream of electrons. The screen is divided into image points or **pixels** (picture elements), each of which can be activated independently. A beam of electrons continuously scans the screen between 30 to 60 times a second. At each scan, it refreshes the image on the screen by activating the appropriate pixels, thus making the image visible by the human eye. Such monitors are also commonly called cathode ray tubes (CRTs) because of their underlying technology.

Laptop computers, as well as many new PCs, use **flat-panel monitors**. These are based on different technologies using LCD or plasma displays. Similar to CRTs, the flat-panel screen is divided into tiny cells, each representing a pixel. But instead of an electron beam, a mesh of horizontal and vertical wires is used to address individual rows and columns of pixels. An electric charge applied to one horizontal and one vertical wire

activates the cell at the intersection of the two wires. In the case of LCDs, an activated cell blocks light passing through it, thus turning dark; in the case of plasma displays, an activated cell emits a burst of light.

The density or **resolution** of the pixels of a display determines the quality of the visible image. A typical 15-inch display has 800 × 600 pixels; a 21-inch display might have 1280 × 1024 pixels.

From the I/O system point of view, monitors may be subdivided into two basic types: character-oriented and graphics-oriented. Each requires a different type of interaction.

Character-oriented displays expect a **stream of characters** and process the stream one character at a time. Each character is either displayed on the screen at the current position of the cursor, or it is interpreted as a **control command**. The meaning of each control character (or a series of such characters) depends on the specific monitor type and the protocol used; typical commands include moving the cursor to the end or beginning of the current line, moving the cursor to the home position (top left of the screen), backspacing the cursor, erasing the current line, performing a "carriage return" (moving cursor to the beginning of the next line), or toggle reverse video mode. The main limitation of character-oriented monitors is that the screen is subdivided into a small and fixed number of rows (lines) and columns (positions within each line), where each position can display only a single character chosen from a fixed set. A typical size of a character-oriented display is 25 lines of 80 characters each.

Figure 11-2a illustrates the principles of character-oriented monitors. It shows a character output buffer in the device controller—a single-character register—that the CPU can write. The register's content, the character *t* in the example, is copied to the display screen at the current cursor position.

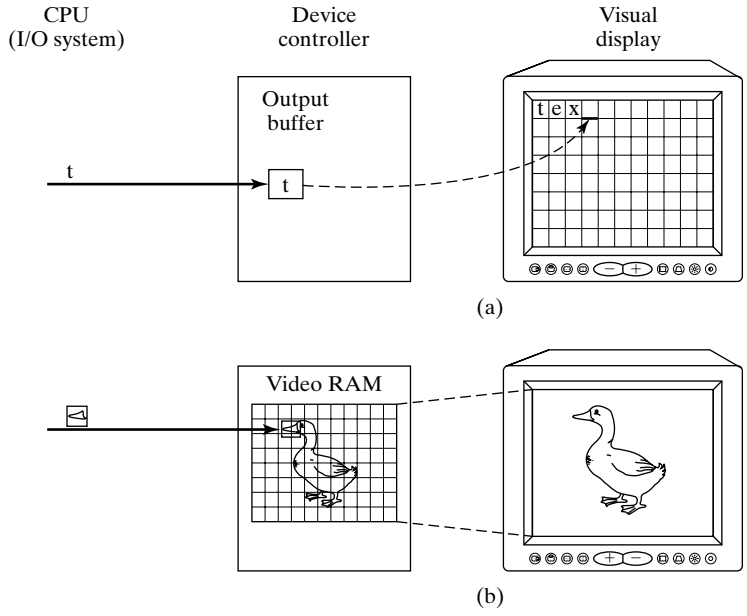

FIGURE 11-2. Display monitors (a) character-oriented; and (b) graphics-oriented.

Graphics-oriented displays use a separate **video RAM** memory to hold a copy of the entire image. Each pixel on the screen has a corresponding location in the video RAM. For a black-and-white screen, a single bit is sufficient to represent a pixel. For color displays, the number of bits determines the number of possible colors for each pixel. One byte allows $2^8 = 256$ different colors, and two bytes allow $2^{16} = 65,536$ different colors. The display hardware continuously reads the contents of the video RAM memory, the so-called **bitmap** of the image, and displays it on the screen.

Graphics-oriented displays have two advantages over character-oriented ones. First, each pixel is represented as a separate cell in the video RAM and hence the bitmap may represent an arbitrary image, restricted only by the display resolution. Second, the cells of the video RAM are randomly accessible by the CPU, and the displayed image may be changed arbitrarily by modifying the corresponding cells. Figure 11-2b illustrates the principles of graphics-oriented displays, showing the CPU updating a portion of the image.

Keyboards

Keyboards have been in existence long before the first computers emerged. In fact, the most common keyboard type still used today, the **QWERTY** keyboard, was designed in the second half of the 19th century for typewriters. Its name derives from the layout of the keys, where the top row of the nonnumeric characters starts with the characters Q, W, E, R, T, Y. Other key layouts have since been found more efficient for computer use and less prone to causing repetitive stress injury (e.g., carpal tunnel syndrome). Nevertheless, the pervasiveness of QWERTY keyboards perpetuates their use, since most people still acquire their typing skills with these keyboards and are reluctant to switch later.

Keyboards depend upon the speed of the human user and are among the slowest input devices. Even the most skilled typist will not be able to exceed a sustained rate of 10 characters per second. At the same time, each typed character generally requires some visible action to be taken, such as displaying the character on the screen. Thus, a keyboard is certainly a character-oriented device. When the user presses a key, the corresponding character is placed into an input buffer within the keyboard controller, from where it may be retrieved by the CPU. From the I/O system point of view, keyboards are similar to character-oriented displays. The main difference is the direction of the data flow; displays consume the characters produced by the CPU, whereas keyboards produce them.

Pointing Devices

Moving the cursor using keyboard keys is not very convenient. The movements are limited to the predefined lines and character positions of displayed text. With graphics-oriented monitors, the user should be able to point to arbitrary positions within the screen, e.g., to switch among applications, to select and move portions of text or images, or to draw line-oriented images. One of the most popular devices to achieve that is the **mouse**, invented in the early 1960s. The most common type, the **optical-mechanical** mouse, uses a small ball that protrudes through a hole at the bottom of the mouse. As the mouse is moved on a flat surface, traction causes the ball to roll. Any change in the horizontal or vertical position, as well as any buttons pressed or released, are detected by the hardware and translated into a stream of bytes. The I/O software retrieves this

data from the input buffer and makes them available to the application software, which determines the corresponding cursor movement to be displayed on the monitor or any other actions resulting from pressing the mouse buttons.

A popular alternative to a mouse is a **trackball**. The hardware and software of a trackball are very similar to those of a mouse. The main difference is that the ball whose movement is being monitored is on top of the device; it is moved directly by the hand, and the device is stationary. Another alternative (or addition) to a mouse is a **joystick**. It is also used to control the cursor, but generally has more control buttons than a mouse or trackball.

Similar to a mouse, the trackball and the joystick report any changes in position or buttons pressed as a stream of events that must extracted from the hardware buffers by the I/O software. Like keyboards, pointing devices are character-oriented. They are also relatively slow, generating streams of data in the range of up to several hundred bytes per second.

11.3.2 Printers and Scanners

Printers convert **soft-copy** output, i.e., information stored in files or displayed on the monitor, into **hard-copy** output, i.e., information printed on paper. Scanners perform the reverse function, by generating digitized soft-copy data from hard copies.

Printers

There are several different technologies used to generate the hard-copy output. At the highest level, we can subdivide all printers into **impact** and **nonimpact** printers. The former include **line printers**, **daisy-wheel printers**, and **dot-matrix printers**. All three work just like typewriters by mechanically imprinting each character through an ink ribbon on the paper.

The simplest nonimpact printer is a **thermal printer**, which uses heated pins to burn images into special heat-sensitive paper. Thermal printers are inexpensive but produce low-quality output. They were used mainly in calculators and fax machines, but most have been displaced by more advanced technologies, including ink-jets and laser printers.

Ink-jet printers, developed by Cannon, have been very popular with PCs due to their relatively low cost, good copy quality, and the ability to produce not only text but arbitrary graphic images. The images are formed by spraying microscopic streams of fast-drying black or color ink on the paper as it passes under the ink jets.

Laser printers, and the related **LCD** and **LED printers**, use the same technology as copy machines. The image is created using laser light (or liquid crystal/light-emitting diodes in the case of LCD/LED printers) on a negatively charged drum. Powder ink (toner) is electrically charged, making it stick to the areas of the image. Next, the image is transferred to paper, which is heated to melt the ink and fuse with the paper.

The two key characteristics of printers are the **quality** of the output and their **speed**. The former is measured in dots per inch (DPI); the latter is given in characters per second or pages per minute. Dot-matrix devices can print several hundred characters per second. Speeds of ink jet and laser printers are in the range of 4 to 20 pages per minute.

Most printers are character-oriented devices; they accept output data in the form of a stream of control and printable characters. The stream is highly device-specific and must be hidden inside the specialized device drivers, generally supplied by the printer vendor.

Scanners

Most scanners use a charged-coupled device array, a tightly packed row of light receptors that can detect variations in light intensity and frequency. As the array passes over the presented image, the detected values are turned into a stream of bytes that the I/O software extracts from the device.

The most popular types of scanners are **flat-bed scanners** and **sheet-fed scanners**. The former are similar to copying machines in that the document to be scanned is placed on a stationary glass surface. Sheet-fed scanners are more like fax machines, where individual sheets of paper must be passed through the machine. Both types of scanner turn each presented page of text or graphics into a bitmap—a matrix of pixels. For black-and-white images, a single bit per pixel is sufficient; for color images, up to 24 bits per pixel are used to represent different colors.

11.3.3 Secondary Storage Devices

Secondary storage is necessary because main memory is volatile and limited in size. If the storage medium is removable, it also may be used to move programs and data between different computers. The most common forms of secondary storage are magnetic or optical disks and magnetic tapes.

Floppy Disks

Floppy disks or **diskettes** use soft magnetic disks to store information. Since their introduction in the late 1960s, floppy disks have been shrinking in size, from 8″ to 5.25″ to the presently most popular size of 3.5″ in diameter.

The information on a floppy is organized into concentric rings or **tracks**. Each track is subdivided into **sectors**, where a sector is the unit of information (number of bytes) that can be read or written with a single operation. The reading and writing is performed by a movable **read/write head** that must be positioned over the track containing the desired sector. As the disk rotates under the read/write head, the contents of the specified sector are accessed.

To simplify the task of the file system or any other application using the disk, the sectors are numbered sequentially from 0 to $n - 1$, where n is the total number sectors comprising the disk. Numbering provides an abstract view of the disk as a linear sequence of sectors, rather than a two-dimensional structure where each sector is addressed by two numbers, a track number and a sector number within the track.

Figure 11-3 illustrates this concept. Figure 11-3a shows a disk surface subdivided into tracks with 18 sectors per track. The sectors are numbered 0 through 17 within each track. Figure 11-3b presents the logical view, where all sectors have been numbered sequentially. Numbering may be done in software, by the lowest level of the I/O system, but more frequently it is done by the disk controller in hardware. Thus, the I/O system only manages the more convenient abstract view of the disks. The hardware can then be optimized for best performance. For example, it may transparently skip defective sectors; it can use different numbers of sectors per track to account for the fact that the physical length of a track depends on its diameter; or it can number the sectors to optimize the seek time and rotational delay.

As an example of the last point, consider the sequential reading of sectors 17 and 18 in Figure 11-3b. It requires moving the read/write head from track 0 to track 1. Since

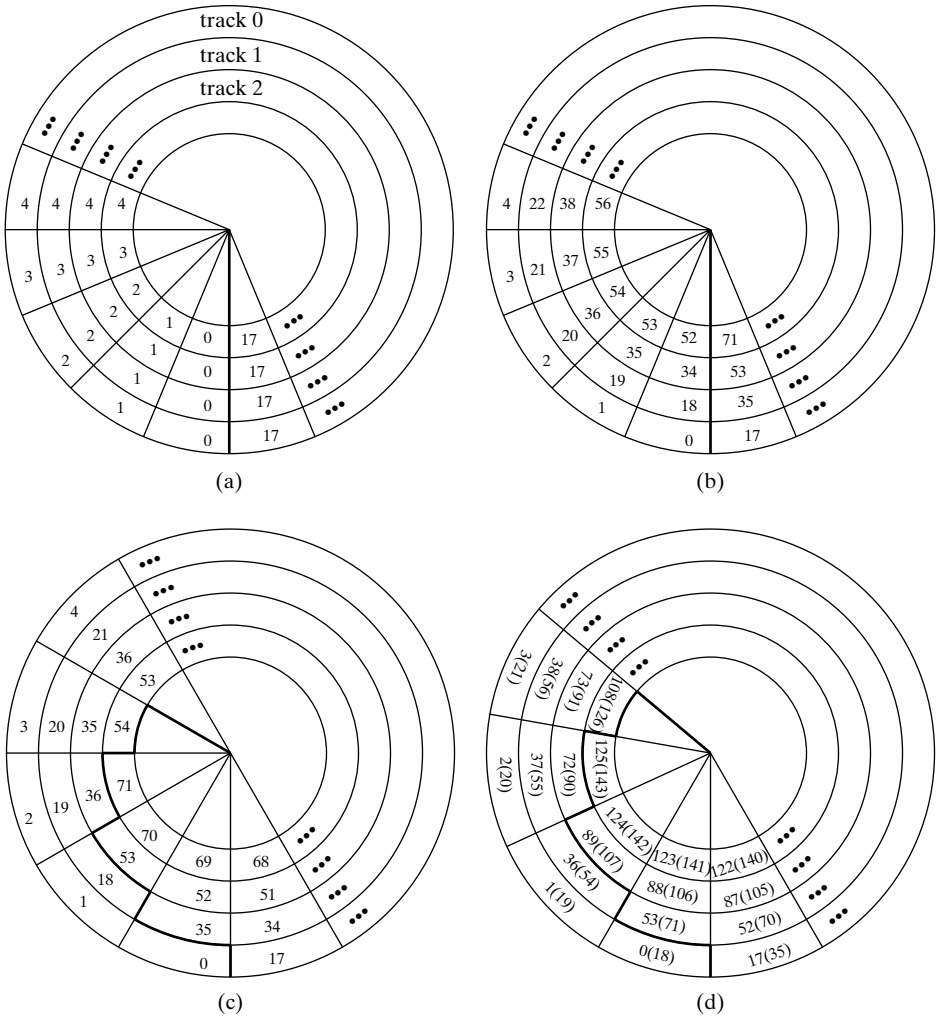

FIGURE 11-3. Numbering of disk sectors (a) physical; (b) logical; (c) with track skew; and (d) with double-sided disk.

this takes some time, sector 18 will already have passed by when the read/write head is in position. The system would then wait for a full rotation of the disk to access sector 18. To eliminate the problem, the sequential numbering of sectors on each track can be offset by one or more sectors to give the read/write arm sufficient time to get into position. Such an offset is termed **track skew**. The numbering in Figure 11-3c defines a track skew of one; this setting would be appropriate if the seek time between adjacent tracks took the same time (or less) as the reading of one sector. In the figure, the read/write head is moving from track 0 to track 1 when sector 35 is passing by, and will be ready to read sector 18 as soon as the head reaches the new track 1.

Some floppy disks use both sides to record information. As a result, there are always two tracks of the same diameter, one on each side of the disk. The number of

sectors that can be accessed without moving the read/write head is essentially double that of the one-sided organization. Thus, when numbering the sectors sequentially, it is important to include sectors of both tracks before moving to the next pair of tracks, to minimize the read/write head movement. Figure 11-3d shows the numbering of the same disk as in Figure 11-3c but assuming both sides to hold information. The first number in each sector gives the logical number of the sector on the top surface of the disk, and the number in parentheses gives the number of the sector on the bottom surface. For example, logical sectors 0-17 are on track 0 (the outermost track) of the top surface, and the next set of sectors 18-35 is on track 0 of the bottom surface. Note that this example also uses a track skew of 1, as in Figure 11-3c.

Hard Disks

Hard disks are similar to floppies in that they also record information on rotating magnetic surfaces. Like floppies, they store this information on concentric tracks, each further subdivided into sectors. The main differences between hard disks and floppies are that the former are *not removable* and can hold much more data. Hard disks generally employ both surfaces of the rotating magnetic platter. Furthermore, many disks use more than one platter stacked on the same rotating axis. Each surface is accessed by its own read/write head, all of which are mounted on the same arm and move in unison. Figure 11-4 presents the basic idea of a multiplatter hard disk. With n double-sided platters, there are $2n$ tracks with the same diameter. Each set of such tracks is called a **cylinder** since the read/write heads are positioned to access one such set without movement and the physical arrangement resembles a hollow cylinder. Thus, a hard disk can be viewed from two different perspectives: as a collection of surfaces, each consisting of different-size tracks or as a collection of cylinders, each consisting of equal-size tracks.

To provide an abstract view of the disk, all sectors of the disk are numbered sequentially. This numbering, generally performed by the controller hardware, is similar

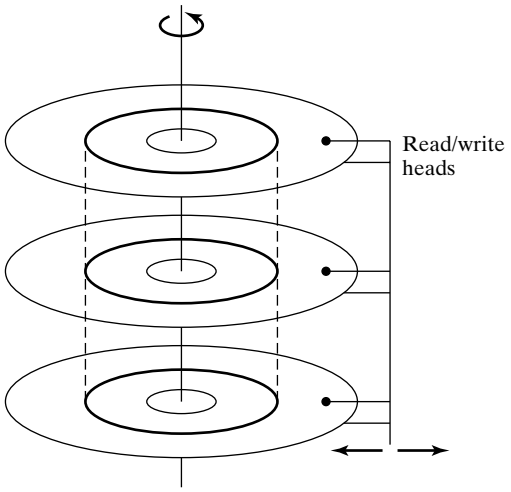

FIGURE 11-4. Organization of a hard disk.

to that of a double-sided floppy; its purpose is to minimize the most time-consuming part of a disk access, which is the seek time between cylinders.

Optical Disks

Optical disks or CDs are different from magnetic disks in a number of important ways. Originally, they were designed to record music in digital form. As a result, information on such disks is not arranged in concentric tracks but along a **continuous spiral**. The individual bits are represented as a sequence of dots along this spiral that either reflect or absorb the light of a laser beam that tracks the spiral. In the early 1980s, the same technology was adapted to store computer data. Such disks are known as **CD-ROMs**. ROM stands for 'read-only-memory,' indicating that information on CD-ROMs, unlike information on magnetic disks, cannot be modified. Consequently, they only can be used for distribution of software or data; this information must be prerecorded on the CD-ROM using a complex process of burning the nonreflecting dots into the aluminum surface of the disk. CD-ROMs have a much higher density of information storage compared with magnetic disks. A double-sided, 4.72-inch CD-ROM can hold more than a gigabyte of digital data—a capacity equal to several hundred floppy disks.

CD-Rs are recordable CD-ROMs. Similar to ordinary CD-ROMs, their contents cannot be modified once written. However, the writing is done using a different technology, one that employs inexpensive CD recorders (CD burners). CR-Rs are also called **WORM**s (write-once/read-many-times). Yet another technology is used for **CD-RWs** that may be read or written multiple times, thus behaving like a magnetic disk. More recent developments in optical disks are **DVDs** (digital versatile disks) and **DVD-RAMs**. They have a different data format that increases their capacity to almost 3 GB per side.

The information on all optical disks is divided into sectors that are similar to magnetic disk sectors and directly accessible by the disk drive. Some drives also support a high-level grouping of sectors, called tracks, to mimic the concentric rings of magnetic disks. Thus, from the I/O software point of view, optical disks are operated using similar principles as magnetic disks: the software specifies the logical sector it wishes to read or write, and the disk controller hardware performs the necessary seek, rotational delay, and data transfer operations.

11.3.4 Performance Characteristics of Disks

Figure 11-5 compares hard disks, floppy disks, and CD-ROM disks in terms of their most important characteristics. Note that the CD-ROM characteristics differ somewhat from those of magnetic disks. This is due to the different technologies used and the intended use of the devices. First, because CD-ROMs store data along a single contiguous spiral, their capacities are given as the number of sectors for the entire surface, rather than broken into two components: sectors per track and tracks per surface. For the same reason, there are no "adjacent" tracks on a CR-ROM, and only the average seek time to an arbitrary sector on the surface is specified. Finally, the rotational speed of a CD-ROM, unlike that of a magnetic disk, is not constant. It varies with the position of the read/write head; the speed is slower when the head is near the center and faster when the head is near the outer edge. The reason for this is that CD-ROM have been designed originally

Characteristic	Floppy Disk	Hard Disk	CD-ROM Disk
Bytes per sector	512	512–4,096	2,048
Sectors per track	9, 15, 18, 36	100–400	333,000
Tracks per surface (number of cylinders)	40, 80, 160	1,000–10,000	(sectors/surface)
Number of surfaces	1–2	2–24	1–2
Seek time (adjacent tracks)	3–5 ms	0.5–1.5 ms	NA
Seek time (average)	30–100 ms	5–12 ms	80–400 ms
Rotational speed	400–700 rpm	3,600–10,000 rpm	(200–530)*k rpm

FIGURE 11-5. Characteristics of floppy and hard disks.

to deliver music, which must be streamed at a constant rate. The basic rotational speed of a CD-ROM ranges from 200 to 530 rpm. The basic speed can be increased k-fold, where k is an even integer between 2 and as much as 40. Such CD-ROM drives are referred to as 2X, 4X, and so on, and their data transfer rates increase by the factor k.

There are two other important performance characteristics of rotating disks, which may be derived from those given in the table: **total storage capacity** and **data transfer rate**. The total capacity of a magnetic disk is obtained by simply multiplying the four values: number of surfaces × number of tracks per surface × number of sectors per track × number of bytes per sector. For example, the capacity of the smallest floppy disk listed in Figure 11-5 is $1 \times 40 \times 9 \times 512 = 184{,}320$ bytes or approximately 184 KB. The typical sizes of floppy disks today are 1.44 MB or 2.88 MB. A fairly large hard disk would have a total storage capacity of $12 \times 10{,}000 \times 300 \times 512 = 18{,}432{,}000{,}000$ bytes or approximately 18.5 GB.

For a CD-ROM, the total capacity is computed as the number of sectors per surface multiplied by the sector length. This is approximately $333{,}000 \times 2048 = 681{,}984$ bytes or 0.66 GB.

The data transfer rate may be interpreted in two different ways. A **peak** data transfer rate is the rate at which the data is streamed to or from the disk once the read/write head has been positioned at the beginning of the sector to be transferred. The peak transfer rate depends directly on the rotational speed of the disk and the number of sectors per track. For example, a disk that rotates at 7200 rpm requires $1/7200 = 0.0001388$ minutes per revolution. This is equivalent to 8.33 milliseconds per revolution. Assuming 300 sectors per track, all 300 sectors will pass under the read/write head in 8.33 milliseconds. Thus, it will take $8.33/300 = 0.028$ milliseconds for one block. Since each block consists of 512 bytes, the peak transfer rate is 512 bytes per 0.028 milliseconds, which corresponds to 18.96 MB per second. The peak data rate is used when determining the total access time to a given disk block or group of blocks, which consists of the seek time, the rotational delay, and the block-transfer time.

For a CD-ROM, the data transfer rate depends on its rotational speed. With a single-speed CR-ROM drive, the data rate is 150 KB/second, and it increases with the speed factor, k, to 300 KB/second for a 2X drive, 600 KB/second for a 2X drive, and so on.

A **sustained** data transfer rate is the rate at which the disk can transfer data continuously. This includes the seek times over multiple cylinders and other overheads in

accessing the data over time. The sustained data transfer rate is an important performance characteristic used by many applications, especially those handling real-time data, to guarantee that they can meet the prescribed demands on delivering data that spans different surfaces and cylinders. The sustained data rate is typically several times lower than the peak data rate.

Magnetic Tapes

Due to their inherently sequential access, low cost, and large storage capacity, magnetic tapes are used mainly for long-term archival storage or for data backup. Magnetic tapes come in several different varieties, packaged as either individual **reels** or **cartridges**. The most common types are **DAT** (digital audio tape) and **DLT** (digital linear tape). DLTs segment the tape into multiple parallel tracks, which results in greater storage capacity and data transfer rate than with DATs.

The storage capacity of DATs ranges from a few hundred kilobytes to several gigabytes, and their data transfer rate is approximately 2 MB/sec. DLTs can store up to 40 GB, and achieve data transfer rates of 2.5 MB/sec.

Information on tapes is generally organized in sequences of bytes called *records* or *blocks*, which can be fixed or variable in length. The tape drive controller supports operations to read or write the next sequential block. I/O software extracts these blocks from or places them into input or output buffers, respectively. Thus, tape blocks are similar to disk sectors for the purpose of data access.

11.3.5 Networks

Networks allow two or more computers to exchange data with one another. There are many types of networks, differentiated by their size, speed, topology, protocols used to exchange data, and the hardware necessary to implement them. From each computer point of view, the network, including all other computers connected to it, is a source or a destination of data, i.e., an I/O device. Like all other devices, the computer has a hardware controller to interface to a network. This **network interface card** (NIC) accepts output data from the I/O software through registers and transmits these to their counterpart controllers in other computers. Similarly, the NIC also accepts input data from other controllers on the network and makes these available to the I/O software in registers.

The protocols used by the different controllers to communicate with each other depend on the network type. For example, an **Ethernet** uses a single common bus to which all controllers are connected. Each can place data on this bus, which all others can receive. A special protocol of rebroadcasting information is implemented by the controllers to solve the problem of collisions—the simultaneous output of data to the bus by two or more controllers. Other common network types are the **token ring** and the **slotted ring**. The former uses a special message (called token) that is circulated among all controllers arranged in a ring. Only the controller currently holding the token may transmit data to another controller on the ring, avoiding any collisions. (A higher-level version of a token ring protocol was developed in Section 3.2.3 to solve the distributed mutual exclusion problem.) Slotted rings continuously circulate data packet frames along the ring, each of which is marked as full or empty. When the controller receives an empty frame, it may fill it with data, mark it full, and pass it on along the ring to its destination controller.

The controller hardware takes care of the lowest layer of the communication protocol. At this level, the controllers have agreed upon the voltage to represent zeros and ones, the physical duration of each bit transmission, how many pins/wires are used to connect the controllers, and other low-level issues. The I/O software handles the high-level tasks, such as transmission errors, parceling of long messages into fixed-size data packets, routing of packets along different communication paths, assembling them at their final destinations, and many other high-level functions necessary to support different applications.

The **telephone network** also can be used by computers to exchange digital data. Since telephone lines were designed to carry continuous voice data, they use analog signals; these are generated as sine waves (the carrier). To transmit digital data, the amplitude, frequency, or phase of the sine wave is varied—a process called **modulation**. A **modem** (for modulator/demodulator) is a device that accepts a stream of bits as input and produces a modulated analog signal, or vice versa. Thus, to send data between two machines through a telephone network, both the sending and the receiving machines must have a modem. The sending modem transforms the digital data into analog signals, and the receiving modem restores the digital information after it has been passed through the telephone lines.

Telephone networks require that a number be dialed and a connection established. Consequently, any data exchange occurs only between two directly connected machines, greatly simplifying the protocols that must be provided by the I/O software. There is no need for packetizing messages, routing and assembly of packets, or any of the other high-level functions of general networks. Instead, the modem can be treated as a character-oriented device that produces and consumes a stream of characters. Thus, from the I/O software point of view, a modem looks more like a printer or a keyboard than a network.

11.4 DEVICE DRIVERS

The tasks of a device driver are to accept commands from the higher-level processes of the I/O system and to interact with the devices it is controlling to carry out these commands. For block-oriented devices, the most common requests are to *read* or *write* a block of data specified by its block number. For character-oriented devices, the most common requests are to *get* or *put* the next sequential character in the data stream. In the case of network drivers, high-level commands, such as those to use sockets, are translated into lower-level operations by the communications protocols. The commands issued to the drivers are to *send a packet* of data to another machine specified by a network address, or to *receive a packet* arriving on the network.

Each of the above requests must be translated further by the driver to the device-specific sequences of low-level operations that are issued to the device controller. The communication between the driver and the controller is accomplished by reading and writing specific **hardware registers** accessible in the device controller. These registers constitute the **software/hardware interface**. Figure 11-6 contains a typical interface for a device controller. The **opcode** register and **operand** registers are written by the driver to describe the operation it wishes the controller to carry out. The number of operands depends on the specific opcode. For example, a request to seek to a specific cylinder on a disk requires the cylinder number as the sole operand; a disk read operation, on the other hand, might require two operands: the track number within the current cylinder and the sector within the track.

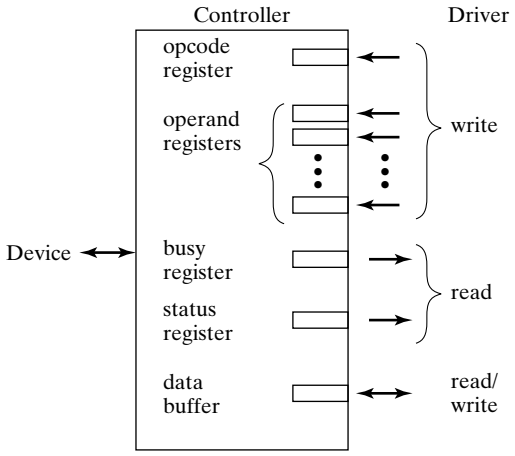

FIGURE 11-6. Device controller interface.

The hardware **data buffer** is used for operations that transfer data to or from the device. For output, the controller transmits the current content of the data buffer to the device. For input, the controller accepts data from the device and places it into the data buffer. The size of the data buffer can vary from a single byte (for character-oriented devices) to thousands of bytes (for fast block-oriented devices).

When the controller detects a new opcode in the register, it begins to work on the specified request. During this time, the **busy** register (which is just a Boolean flag) is set to true, indicating that the controller is busy and cannot accept any new requests. The outcome of a completed operation is reported in the **status** register. Possible outcomes, aside from normal completion, include a wide variety of potential problems or failures depending on the device type and the operation requested. The status register is usually read by the driver after the controller has indicated completion of its work by resetting the *busy* flag. If the operation completed successfully, the driver may issue the next operation, or, in the case of an input operation, it may access the data in the data buffer. If the operation failed, the driver must analyze the cause of the fault and either attempt to correct it (e.g., by retrying an operation that failed as the result of a transient fault), or report failure to the higher-level process that issued the request.

The controller shown in Figure 11-6 is quite complex. It is representative of fast block-oriented device controllers, such as those used for disks. The controllers for character-oriented devices generally have simpler interfaces, consisting of an input and/or an output character buffer, but no opcode or operand registers. An input device places individual characters into the input buffer, which the CPU retrieves using a simple "handshake" protocol of busy/ready signals or using interrupts. A stream of characters from the CPU to an output device is handled in an analogous fashion.

Many computers come from the factory already equipped with a set of generic controllers that may connect to a variety of devices. The most common controllers are the **serial port** and the **parallel port**. Both transmit individual characters to or from a device; the former sends the data one bit at time along a single wire, and the latter uses multiple parallel wires to send each character in one step.

11.4.1 Memory-Mapped Versus Explicit Device Interfaces

A driver specifies each I/O request to a controller using the register interface. One way to accomplish this is to provide *special I/O instructions* to read and write the controller's registers. Another method is to extend the address space of the main memory and to *map the controller registers* into the new memory addresses. Figure 11-7 illustrates the two approaches. In the first case (Fig. 11-7a), the CPU must distinguish between main memory and device addresses, using a different instruction format for each. For example, to store the content of a given CPU register (*cpu_reg*) into a given memory location (*k*), the following instruction could be used:

```
store cpu_reg, k
```

In contrast, to copy the content of the same CPU register into a controller's register, an instruction with a different opcode and format would be used, for example:

```
io_store cpu_reg, dev_no, dev_reg
```

where *dev_no* specifies the device (i.e., its controller address), and *dev_reg* names the register within that controller.

The clear disadvantage of this approach is that two different types of instructions must be used for main memory addressing and device addressing. The second method, known as **memory-mapped I/O**, allows the same instruction format to be used for both

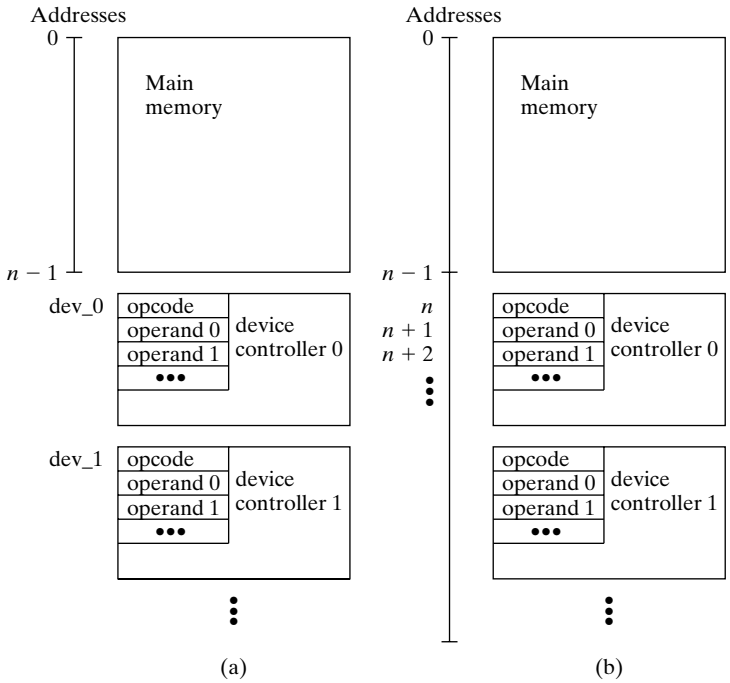

FIGURE 11-7. Forms of device addressing: (a) explicit; and (b) memory-mapped.

main memory and devices. For example, in (Fig. 11-7b), the instruction:

```
store cpu_reg, k
```

where k ranges from 0 to $n-1$, would store the CPU register content into main memory locations as before. But with $k \geq n$, the same instruction stores the CPU register into one of the controller registers. For example, the opcode register of device 0 is mapped to address n in the figure. The instruction:

```
store   cpu_reg, n
```

stores the CPU register content into register 0 of device 0. This uniform view of main memory and devices greatly simplifies I/O programming. It is analogous to the simplification obtained by memory-mapped files, which were discussed in Section 10.3.2.

Having defined the register-based interface between the driver software and the controller hardware, we can now address two important issues, leading, in turn, to several different forms of I/O processing:

1. After an operation has been issued and the controller becomes busy with the I/O, how should operation completion be detected?
2. In the case of operations that input or output data, what should move the data between the controller data buffer and main memory?

11.4.2 Programmed Input/Output with Polling

In the simplest form of I/O processing, the CPU has complete responsibility for the job. That means, it explicitly detects when the device has completed an assigned operation and, in the case of a data I/O operation, it moves the data between the controller register and main memory. The first task is accomplished by repeatedly reading and testing the controller's busy flag—a task commonly referred to as **polling**. Once the flag becomes false, the CPU may access the data buffer. In the case of an input operation, it would copy the data to main memory; for an output operation, it would send data from memory to the buffer, prior to issuing the corresponding output operation.

Figure 11-8 illustrates the protocol for programmed I/O with polling for an input operation. It includes the following steps:

1. The CPU writes the operands required for the input operation into the appropriate operand registers. The number and type of operand required depends on the controller's sophistication and the operation type. A simple disk controller may require the track number and sector number to be specified for an input or output operation, assuming that the read/write head of the disk has already been positioned to the correct cylinder with a seek operation. A more advanced controller might accept a logical disk block number, from which it automatically derives the cylinder, track, and sector numbers, and performs the seek operation, if necessary. In contrast, getting the next character from a keyboard or another inherently sequential device requires no operands to be specified.

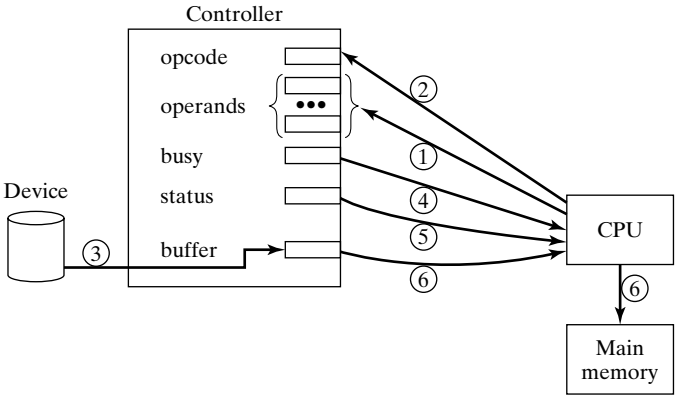

FIGURE 11-8. Programmed I/O with polling.

2. The CPU writes the opcode for the input operation, which triggers the controller to execute the operation. At that time, the busy flag is raised, indicating that the controller is busy and will ignore any other operations issued to it during that time.
3. The controller interacts with the device using the appropriate device-specific hardware protocols to carry out the requested operation. It transfers the input data extracted from the device to the data buffer.
4. While the controller/device are working, the CPU is polling the controller by repeatedly reading and testing the busy flag. (This is just another form of busy-waiting, as discussed in earlier chapters.)
5. Once the operation completes, the CPU reads the status register to detect any possible problems reported by the controller.
6. Provided no errors have been reported, the CPU copies the contents of the data buffer to main memory, where it is made available to the requesting process.

The protocol followed for an output operation is analogous. The main difference is that the driver, having detected that the device is not busy, places the data to be output into the controller data buffer. It then writes the necessary operands (as in step 1 above) and opcode (as in step 2), carries out the operation by transferring the data from the buffer to the device (step 3), polls the device for completion (step 4), and examines the status of the operation (step 5).

Let us examine the I/O programming necessary to implement this polling scheme. Assume that a sequence of characters is to be read from a sequential device (e.g., a modem) and written to another sequential device (e.g., a printer). To accomplish this I/O, the CPU repeatedly executes the above protocol of copying data between main memory and the controller buffer, writing the controller registers, and testing the busy flag and the status register. The following pseudocode describes the operation performed by the driver at a slightly higher level of abstraction than the register-based protocol:

```
Input:
     i = 0;
     do {
```

```
            write_reg(opcode, read);
            while (busy_flag == true) {...??...};
            mm_in_area[i] = data_buffer;
            increment i;
            compute;
      } while (data_available)

Output:
      i = 0;
      do {
            compute;
            data_buffer = mm_out_area[i];
            increment i;
            write_reg(opcode, write);
            while (busy_flag == true) {...??...};
      } while (data_available)
```

The *write_reg* instruction represents the writing of the necessary registers; in this case, the *opcode* register that initiates the input operation. Thereafter, the driver program polls the busy flag by repeatedly executing the while-loop until *busy_flag* turns to false. (The meaning of the statement labeled "??" will be explained shortly.) Testing of the status register has been omitted in the code for simplicity. Assuming that the operation was successful, the driver copies the current character from the controller data buffer to main memory. The main memory area (i.e., the software buffer) is represented as a character array, *mm_in_area[]*, and *i* is the position to store the current character; this position is incremented each time a character has been deposited. The final statement of the loop, named *compute*, stands for any subsequent computation, such as processing the just-received character. The operations for the output loop are analogous.

The problem with polling is that it uses the CPU for both detection of the I/O completion and for moving of all data between the controller and main memory. Both tasks place a heavy burden on the CPU. The need for polling is especially bothersome in the case of slow character-oriented devices. The CPU may spend much of its time busy-waiting, i.e., executing the while-loop waiting for the busy flag to turn false.

There are several ways to address the looping problem, none of which is very satisfactory. One can simply accept the situation, letting the CPU idle while waiting for the I/O completion. This corresponds to turning the statement labeled "??" in the above pseudocode into a "no-op," i.e., an empty body of the while-loop. This waste of significant CPU time is not acceptable in most cases.

A better approach is to poll less frequently by performing other work during each iteration of the wait-loop. That means, the statement "??" can be replaced by another computation that does not need the current input or output. This improves CPU utilization by overlapping its execution with the I/O processing. The difficulty with this strategy is that a driver usually does not have anything else to do while waiting for an I/O operation to complete. Hence, the technique is only feasible in systems where the low-level I/O operations are visible at the application level. If an application is able to initiate an input operation (by writing the controller registers) and has access to the busy flag, it can explicitly interleave I/O with computing to maximize the performance of both.

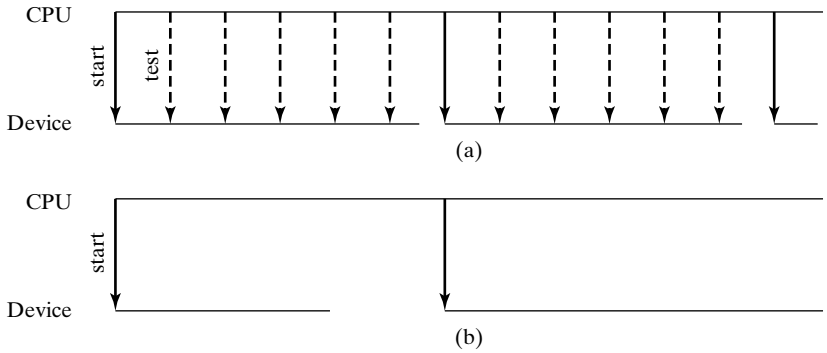

FIGURE 11-9. Frequency of polling: (a) too frequent; and (b) too sparse.

Unfortunately, such interleaving is extremely difficult to implement. The programs must be carefully organized with respect to the spacing of I/O commands and compute-only sections of code if a reasonable overlap of the two is to be obtained. If the busy flag is tested too infrequently, the device will be underutilized; testing it too frequently, on the other hand, results in wasted CPU time.

The diagram in Figure 11-9 illustrates the dilemma. Horizontal lines represent the time during which the CPU and the device are busy; each solid vertical arrow denotes the testing and subsequent starting of the device by writing the opcode register, and each dashed vertical arrow indicates the testing of the busy flag. In Figure 11-9a, the other extreme, the polls are issued too frequently, resulting in wasted CPU time. In Figure 11-9b, the polls are too sparse, resulting in device underutilization. Finding the right balance, especially when the busy times of the device may vary, is very difficult. The approach is also unsuitable for time-sharing environments, where the CPU is transparently switched between different processes and loses control over the real time in which the devices operate.

The third possible way to reduce CPU overhead caused by polling is to have a process voluntarily give up the CPU after issuing an I/O instruction. Other processes can use the CPU when the I/O operation is in progress. When the waiting process is restarted again as the result of normal process scheduling, it can test the busy flag. If the device is still busy, it again gives up the CPU; otherwise, it continues to execute. This guarantees good CPU utilization (provided there are other processes to run), but the device utilization is likely to be poor. That is because the process issuing the I/O command is not restarted *immediately* when the I/O completes. The busy flag is not tested until the process is resumed by the scheduler, resulting in delays of unpredictable length. Such delays are not only wasteful but also may lead to data loss. Many devices (e.g., modems) produce data at a fixed rate; if the data is not consumed in a timely manner, it is simply overwritten with new data.

11.4.3 Programmed Input/Output with Interrupts

From the above discussion of polling, we conclude that it is very difficult to find useful work of appropriate size that a process could perform when waiting for an I/O completion. A better solution is to suspend the process when the I/O is in progress and let other processes use the CPU in the meantime. In a multiprogramming environment,

there are usually enough processes ready to run. The waiting process, however, should be restarted as soon as the I/O operation completes, to optimize the use of I/O devices. This can be achieved using **interrupts.** Recall from Sections 1.2.2 and 4.6 that an interrupt is a signal that causes the CPU to suspend execution of the current computation and transfer control to a special OS kernel code that determines the reason for the interrupt and invokes an appropriate routine (the interrupt handler) to address the event that triggered the interrupt.

When a device controller has completed an I/O operation, it triggers an interrupt. At this point, the next I/O instruction, if any, can be issued without delay. Thus, it is not necessary to continuously monitor the busy flag, which allows the CPU to devote its full attention to running other processes. Figure 11-10 sketches one common protocol for programmed I/O with interrupts for an input operation. The following steps are executed:

1. The CPU writes the operands required for the input operation into the appropriate registers in the controller.
2. The CPU writes the opcode for the input operation, which triggers the controller to start the operation. As in the case of polling, the busy flag is raised. This need not be tested explicitly to detect the I/O completion. Rather, it is only used initially to ascertain that the device is not busy (e.g., serving a request issued by another process) and is able to accept a new command. Once the operation has been initiated, the process blocks itself, giving up the CPU.
3. The device controller interacts with the device to carry out the current operation. In the case of an input operation, it transfers the data from the device to its data buffer. In the meantime, other processes may be running on the CPU.
4. When the operation is complete, the controller issues an interrupt. This causes the currently running process to be suspended and the CPU branches to the interrupt handler. This analyzes the cause of the interrupt and resumes the process waiting for the just-completed I/O operation.
5. The resumed process verifies that the operation has completed successfully by examining the status register.

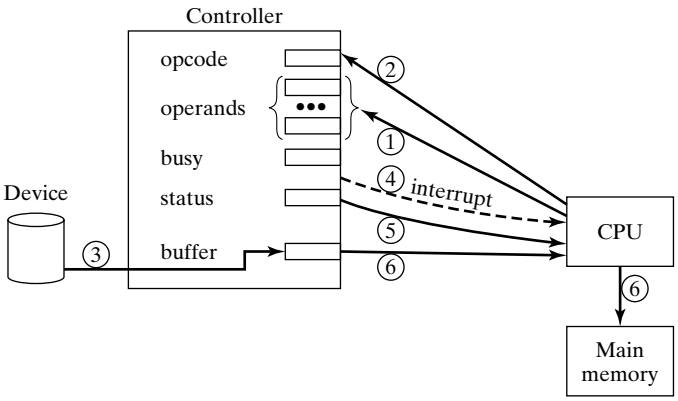

FIGURE 11-10. Programmed I/O with interrupts.

6. Assuming no errors have been detected, the process copies the data from the controller data buffer into main memory.

Note that the main conceptual difference between the two I/O protocols is in step 4. In the case of polling, the process requesting the I/O operation is also in charge of detecting its completion. In the case of interrupts, the requesting process is blocked, and other components of the OS, notably the interrupt handler, must be involved in detecting I/O completion.

Let us now examine the style of interrupt-driven I/O programming and compare it to the previous scheme of explicit polling. Assume a task that reads and writes a simple sequence of characters from/to a sequential device, as in the example of the last section. The following pseudocode describes the operations performed by the driver. This is essentially the interrupt service routine for the I/O interrupt.

```
Input:
     i = 0;
     do {
          write_reg(opcode, read);
          block to wait for interrupt;
          mm_in_area[i] = data_buffer;
          increment i;
          compute;
     } while (data_available)

Output:
     i = 0;
     do {
          compute;
          data_buffer = mm_out_area[i];
          increment i;
          write_reg(opcode, write);
          block to wait for interrupt;
     } while (data_available)
```

The code is similar to that of polling, with the main difference of no busy-wait loop testing the busy flag. Instead, the process blocks itself by giving up the CPU after initiating the I/O operation using the *write_reg* operation. When the process restarts as the result of the I/O completion interrupt, the data is ready to be copied from the data buffer (in the case of input) or to the data buffer (in the case of output). The *compute* operation in each of the loop again represents some processing that must be done for each character of input or output.

EXAMPLE: Keyboard Input

The following pseudocode implements the typical functions of a keyboard driver:

```
Keyboard_Input:
     i = 0;
     do {
```

```
            block to wait for interrupt;
            mm_in_area[i] = data_buffer;
            increment i;
            compute(mm_in_area[]);
        } while (data_buffer ≠ ENTER)
```

Note that this code is very similar to the interrupt-driven input sequence shown above, with only a few minor differences. First, there is no *write_reg* operation to start the input of a character. That is because the keyboard generates a character in its data buffer whenever a key is pressed or released by the user, without any explicit prompting. The driver's task is to copy this character into the appropriate area in main memory. The loop is repeated until the character in the buffer is the ENTER key (also called RETURN or CR, for carriage return, on some keyboards). This signals that the current line is complete and should be made available to the calling process.

The meaning of the statement *compute(mm_in_area[])* depends on the type of input desired by the process. In **character-oriented** input, all characters, including special characters, such as CONTROL, SHIFT, ALT, DELETE, or BACKSPACE, are copied to the memory area, and it is up to the process to interpret their meaning. This mode of input is usually called **raw input** and is of interest to sophisticated text editors that are able to interpret control characters according to the user's definitions. Most programs, however, do not need to see all characters. For example, when the user mistypes a character, and corrects it by pressing BACKSPACE or DELETE followed by the correct character, the process should see only the final complete line at the point when ENTER is pressed, rather than all the intermediate versions resulting from typing errors. Thus, it is up to the driver to perform all the intraline editing by interpreting special characters used for that purpose. Such **line-oriented** input mode is usually referred to as **canonical** or **cooked input**.

The timing diagram of Figure 11-11 shows the interactions between the CPU software and an I/O device for typical interrupt-driven input. Assume that a currently running user process issues a *read* operation. This is passed to the I/O system and eventually causes the appropriate device driver to be invoked to process the operation. The driver writes the necessary controller registers and starts the operation. At some time after the *read* and before or shortly after the device becomes busy, the current process is blocked. The process scheduler is invoked and selects the next ready process to run. As long as the device continues to be busy, other ready processes may be timeshared on the CPU. Eventually, the I/O operation completes and generates an interrupt, thereby suspending the currently running process and starting the interrupt handler. Determining that the interrupt was caused by I/O completion, the interrupt handler returns control to the driver in charge of the interrupting controller. The driver now analyzes the status of the just-completed read operation, and, if successful, transfers the data from the controller buffer to main memory. If more data are to be input, the driver immediately restarts the device, minimizing its idle time. When this is done, the process scheduler is invoked again to determine which process should continue executing. In the diagram, we assume resumption of the original user process, which now has the previously requested input data in its main memory area available for access.

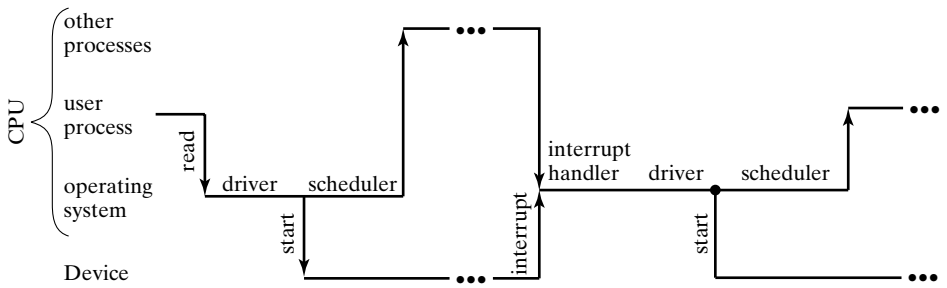

FIGURE 11-11. Timing of interrupt-driven I/O.

CASE STUDY: I/O PROCESSING IN UNIX

Device drivers in UNIX are divided into two parts, called the **top half** and **bottom half**, respectively. The top half is invoked **synchronously**, whenever a process requests an I/O operation. In Figure 11-11, the *read* operations would invoke the top half of the driver, which starts the device and terminates. The bottom half is invoked **asynchronously** by the interrupt when the device completes. The bottom half copies the data from/to the hardware buffer. Then, depending on the type of operation, it either terminates or invokes a portion of the top half, which restarts the device with the next operation.

In the case of terminal input (keyboard), the two halves are completely independent. The bottom half is invoked whenever a key is pressed on the keyboard (or mouse). It copies the corresponding character from the hardware buffer into a FIFO buffer, called C-list, in memory (see Section 11.5.1). A process retrieves characters from this buffer by invoking the top half of the driver. This automatically blocks the process when the FIFO buffer is empty.

With interrupts, the completion of an I/O operation is detected immediately and automatically. Thus, a process must not worry about testing the device flag at various intervals to optimize CPU and I/O device use. However, Figure 11-11 also illustrates that the flexibility gained by interrupt-driven processing over polling is not free of cost. The overhead caused by an interrupt is quite significant, involving the invocation of the interrupt handler and other OS routines. These can take thousands of CPU instructions to execute. Testing the busy flag, in contrast, can be done in just two machine instructions—a register read followed by a conditional branch. Nevertheless, polling is effective in only very special circumstances, e.g., with certain dedicated embedded systems or when a process must wait nondeterministically for multiple devices to complete. Virtually all general-purpose computers, including PCs, employ interrupts to manage I/O and other operations.

11.4.4 Direct Memory Access

With programmed I/O, either through polling or interrupts, the CPU carries the burden of physically moving all data between main memory and the controller data buffers. This

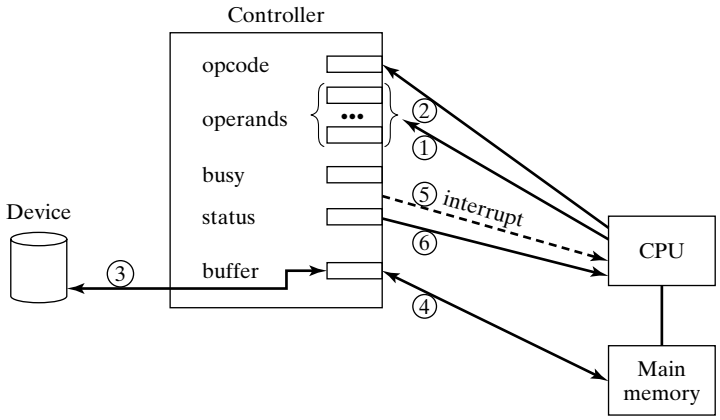

FIGURE 11-12. Direct memory access I/O (DMA).

is acceptable with slow, character-oriented devices, such as keyboards or printers, since the CPU is capable of executing thousands of instructions before a device completes the transfer of a single character. In the case of fast devices, however, the CPU overhead in initiating and monitoring each individual transfer of data between a device and main memory is too high. To alleviate this problem, **direct memory access** (DMA) hardware can be added to the system. It permits a device controller to transfer data directly to or from main memory. Using DMA, the CPU only commands the controller to perform the transfer of a block of data; the actual operation is carried out directly by the DMA controller. I/O completion can be detected, in principle, with polling as in the case of programmed I/O. However, interrupts are much more effective since the main objective of DMA is to liberate the CPU from as much overhead as possible.

The steps taken by the CPU when interacting with a DMA controller are marked in Figure 11-12. Although seemingly not much different than Figure 11-10, there are several significant changes in detail.

1. The CPU writes the operands relevant to the current I/O operation to the controller operand registers. This is generally more complex than with programmed I/O, since the DMA controller has more autonomy in executing I/O operations than a programmed one. In particular, the operands must include the starting location in main memory from or to which data should be transferred, and the number of bytes to transfer.
2. The CPU stores the opcode into the controller opcode register. This initiates the controller, which raises its busy flag to indicate that it will not accept any other commands in the meantime. When the controller is busy, the CPU is free to execute other computations.
3. The controller interacts with the device to carry out the data transfer to or from its data buffer.
4. The controller copies the data between its data buffer and the area in main memory specified as one of the operands. The steps 3 and 4 may be performed repeatedly to transfer a series of characters or blocks between the device and main memory.

5. When the operation is completed, the controller resets its busy flag and issues an interrupt to the CPU, indicating that it is ready for the next command.
6. The CPU reads and tests the controller status register to determine if the operation has been carried out successfully.

The CPU's involvement in I/O processing has been greatly reduced. Consider the task of reading or writing a stream of characters into/from main memory. The following pseudocode would accomplish this task:

```
Input or Output:
          write_reg(mm_buf, m);
          write_reg(count, n);
          write_reg(opcode, read/write);
          block to wait for interrupt;
```

Compare this simple code with the analogous code using programmed I/O (with interrupts), as shown in Section 11.4.3. Using the latter, the CPU must execute a loop, which inputs one character during each iteration. With DMA, the CPU only must write the opcode register and the appropriate operands, which include the starting address in main memory (m) and the number of characters to read or write (n). Thereafter, the CPU is free to execute other computations. It will be interrupted only after the controller has transferred *all n* characters into main memory (or has failed in some way).

One problem with direct memory access is that the DMA controller is interfering with the normal operation of the CPU—both compete for access to main memory. When the controller is in the process of reading or writing data in main memory, the CPU is momentarily delayed; this interference is known as **cycle stealing**, since the DMA has 'stolen' execution cycles from the CPU. Given that the use of DMA does not increase the total number of memory accesses in any way and that the CPU is frequently accessing data in its internal registers or in cache, thus bypassing main memory, the resulting conflicts are limited and well worth the gained benefits, i.e., a dramatic reduction in the CPU's involvement in I/O processing.

Although DMA liberates the CPU from directly performing data transfers, there is still a significant amount of work to be done by the CPU for each data transfer. In particular, it must analyze the status of each interrupting device to detect possible errors, attempt to correct or recover from discovered ones, and perform various device-specific code conversions and formatting functions. All these tasks may be delegated usefully to a specialized **I/O processor** or **channel**. This specialized device may be viewed as a more sophisticated form of a DMA controller, responsible for managing several devices simultaneously and supervising the data transfers between each of these devices and main memory.

Unlike a simple DMA controller, which only interprets the commands given to it via its register interface, an I/O processor has its own memory from which it can execute programs. Such programs can be complex sequences of specialized I/O instructions, logical and arithmetic operations, and control statements. Each I/O program is prepared and stored in the I/O processor executable memory by the CPU. The CPU then initiates the I/O processor, which executes its current program and interrupts the CPU only when it has completed its task. In this way, the I/O processor can be made responsible for

transferring complex sequences of data, including all necessary error corrections or data conversions, without any assistance by the CPU.

11.5 DEVICE MANAGEMENT

Much of the I/O system is highly device-dependent, particularly individual device drivers. However, there are numerous functions that are common to many devices and that may be performed at a higher, device-independent level. This section discusses the general principles of several such functions, including buffering and caching, error handling, and device scheduling and sharing.

11.5.1 Buffering and Caching

A buffer is an area of storage, such as a dedicated hardware register or a portion of main memory, that is used to hold data placed there by a **producer** process until it is copied by a **consumer** process. We can distinguish two types of buffers, based on their organization and use: a FIFO buffer and a direct-access buffer.

Figure 11-13a shows the first type, a **FIFO buffer**. The key reason for using a FIFO buffer is to *decouple* the producer and consumer processes in time. Without a buffer, the two processes must tightly synchronize their executions such that the consumer is ready to receive the data while it is being generated by the producer. The use of a buffer allows concurrent asynchronous execution of the producer and the consumer.

Figure 11-13b shows a **direct-access buffer**, which is also referred to as a **buffer pool** or **buffer cache**. A direct-access buffer also decouples the execution of the producer and consumer. The key reason for its use is to avoid producing the same data *repeatedly* when it is accessed multiple times by the consumer. For example, when the file system (the consumer) accesses the same file block multiple times, the I/O system (the producer) should not have to read the block from disk repeatedly. Instead, it uses the buffer as a cache to keep the most recently accessed blocks ready for future use.

There are two other important reasons for using buffers, both FIFO and direct-access. The first is to handle any mismatches in **granularity** of the data exchanged between the producer and the consumer. If the producer's data granularity is smaller

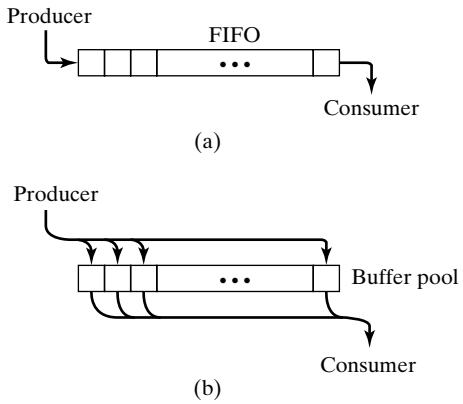

FIGURE 11-13. Buffer types: (a) FIFO buffer; and (b) direct-access buffer.

than the consumer's, the producer can deposit several of its data units in a buffer to make one large unit for the consumer. Conversely, if the producer's granularity is larger, the consumer can remove each data item from the buffer in several smaller units.

The second reason for buffering is to provide memory space that is not part of the producer or the consumer. This allows the consumer to be swapped out while it is waiting for data to be produced. Conversely, the producer can be swapped out while data is being consumed from the buffer. Both cases improve the use of main memory.

The number and organization of the slots within a buffer (characters or blocks, depending on the application) is critical for performance. Let us examine the common choices and their trade offs.

Single Buffer

A single buffer allows the producer to output data without waiting for the consumer to be ready. Such asynchronous transfer of data between processes is important for performance. However, the producer and consumer processes must still coordinate their operations to ensure the buffer is full when the consumer accesses it and "empty" (i.e., its contents have already been copied) before it is overwritten by new data.

A single buffer is employed typically with simple device controllers, as discussed in Sections 11.4.2 and 11.4.3. In the case of input devices, the controller is the producer, and the device driver is the consumer. In the case of output, the producer/consumer roles are reversed. In both situations, the data are exchanged one character or block at a time via the controller data buffer.

One drawback of a single buffer scheme—and a source of unnecessary loss of concurrency—is that the producer is idle when the consumer is copying the buffer and, conversely, the consumer is idle when the buffer is being filled.

Figure 11-14a shows the timeline for the loop with a single buffer. This corresponds to the input sequence using polling (Section 11.4.2), where the device is the producer and the CPU (the device driver) is the consumer. The CPU starts the device, and it repeatedly polls its busy flag. When the I/O is completed, the buffer is copied to main memory. The CPU is idle when the buffer is being filled, and the device is idle when the buffer is being copied to memory.

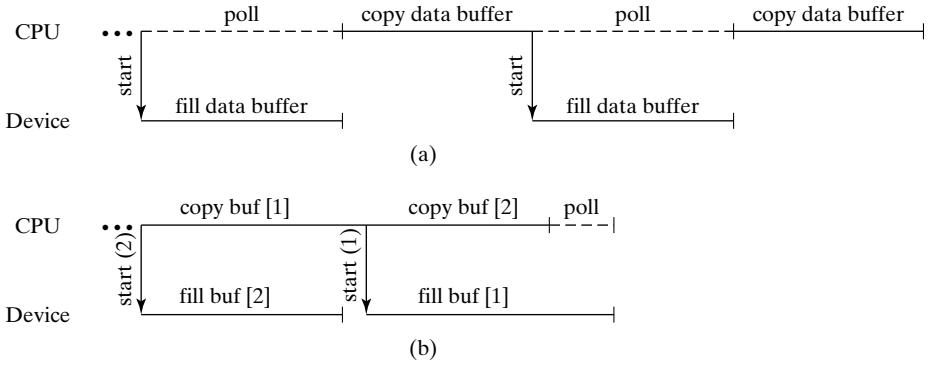

FIGURE 11-14. Buffering: (a) single buffer; and (b) buffer swapping.

Buffer Swapping

We can reduce the idle time resulting from the exclusive use of the buffer by employing two separate buffers, say *buf[1]* and *buf[2]*, in the controller. When the controller is filling *buf[1]*, the CPU may be copying *buf[2]*, and vice versa. The roles of the buffers are reversed at the completion of each I/O operation. The following code to input a sequence of characters illustrates the implementation of this **buffer-swapping** technique.

```
Input:
     i = 0;
     b = 1;
     write_reg(mm_buf, buf[2]);
     write_reg(opcode, read);
     do {
          while (busy_flag == true) ; /* busy-wait */
          write_reg(mm_buf, buf[b]);
          write_reg(opcode, read);
          if (b == 1) b = 2; else b = 1;
          mm_in_area[i] = buf[b];
          increment i;
          compute;
     } while (data_available)
```

The variable *b* indicates which of the two buffers is being used; initially, it is set to 1. This is followed by initial *write_reg* operations, which write the necessary opcode and operand registers. One of the operands is the buffer number (1 or 2) to be used for the next operation. The initial *write_reg* operations start the filling of *buf[2]*. The repeat loop alternates between the two buffers: Whenever it issues an operation with *buf[1]* as the operand, it changes *b* to 2, and vice versa. That way, the subsequent main memory copy always uses the other buffer, i.e., the one not being currently filled.

Figure 11-14b illustrates the resulting overlap between the CPU by the device and the input device. The time to input each character during each iteration is the maximum of the memory copy operation and the device input operation. Assuming these times are comparable, the technique of buffer swapping eliminates much of the idle time observed in Figure 11-14a.

Circular Buffer

Buffer swapping improves performance by increasing the overlap between the CPU and I/O, and, in fact, is optimal under the ideal condition where the data are produced and consumed at a constant rate. Unfortunately, timing sequences in real-world situations are rarely that simple. Programs usually contain bursts of I/O activity followed by periods of computing. Also, the times for consuming (copying) the data may depend on the data type and other circumstances, and vary from call to call. Similarly, the times that the controller is busy may vary because different physical records may be accessed on each call. For example, accessing a disk block that requires a seek operation is much slower than accessing a block on the current cylinder. Finally, in a multiprogramming environment, the speed with which a given process is executing and the current availability of devices

are unpredictable. As a result, a user process frequently waits for I/O to complete, and a device frequently remains idle as a process copies the buffer.

The solution to this problem is to add more buffers to the system to further increase the asynchrony between the producer and the consumer. With n buffers, the producer can produce up to n data items before it must block and wait for the consumer to empty some or all of them. Conversely, when all n buffers are full, the consumer can execute up to n iterations before it must block to wait for the producer.

This solution corresponds to those given for the general bounded buffer problem in Chapters 2 and 3. All of the latter programs assumed a buffer capable of holding n data items and guaranteed that the producer would run up to n steps ahead of the consumer, without any loss of data. The buffer organization is referred to as a **circular buffer** because the producer and consumer both access the buffer sequentially modulo n.

EXAMPLE: Implementing a Circular Buffer

The circular buffer concept is outlined in Figure 4-2a. Whenever the producer continues, it fills the first empty buffer pointed to by the *front* pointer. Similarly, the next buffer accessed by the consumer is the one pointed to by the *read* pointer.

The following code demonstrates the use of the circular buffer monitor (Section 3.1.1) in an I/O application. The buffer resides between a user on file system process (the consumer) performing input and the driver of the corresponding input device (the producer).

```
User Process:
       do {
            buffer.remove(data);
            compute(data);
       } while (data_available)

Driver:
       do {
            write_reg(...);
            block to wait for interrupt;
            buffer.deposit(data_buffer);
       } while (data_available)
```

The user process repeatedly calls upon the buffer monitor to obtain the next data item and processes it using the subsequent *compute* statement. If it runs too far ahead of the driver so that all buffers become empty, the monitor guarantees that the user process blocks as part of the call; it resumes automatically when at least one buffer is filled.

The driver repeatedly obtains data from the input device by writing the appropriate registers of the device controller and blocking itself to await the I/O completion. When awakened by the interrupt, it places the new data copied from the controller *data buffer* into the circular buffer by calling the *deposit* function of the buffer monitor. In case the user process falls behind in consuming data and the circular buffer is completely full, the monitor blocks the driver as part of the call. It wakes it up again automatically when at least one buffer is free. Thus, the driver can be blocked for *two* different reasons: to wait for the device controller or to wait for the user process.

Buffer Queue

The circular buffer can be implemented efficiently as an array. But its main limitation is its fixed size, n, which is a very important parameter. If n is too large, memory space is wasted. If n is too small, the produce or the consumer process are blocked frequently. In some situations, no blocking of the consumer may be possible. For example, a modem delivering data cannot be stopped; if the data is not consumed at the rate is it produced, it is lost. Similarly, a user typing at a keyboard cannot be told to stop; the system must accept all characters as they are being delivered.

The problem can be solved by implementing an unbounded buffer in the form of a buffer queue, similar to that of Figure 4-2b. For each new data item to be inserted, the producer creates a new list element and inserts it at the front of the queue. The consumer removes elements from the rear of the queue. This allows the producer to run arbitrarily far ahead of the consumer. The main drawback of such a buffer queue is efficiency. Manipulating a linked list is time-consuming, because every operation requires dynamic memory management.

CASE STUDY: C-LISTS IN UNIX

UNIX implements a version of a buffer queue, referred to as a **C-list**, which is a compromise between a linked list and an array implementation. This is used for character input and output. The idea is to organize *arrays of characters*, rather than individual characters, in the form of a linked list. Figure 11-15 illustrates this concept. It shows a buffer queue containing 104 characters spread over three arrays of 64 characters each. The consumer (e.g., a user process) reads characters using and incrementing the rear pointer. When it reaches the end of the current array, it unlinks it from the queue and continues with the next array. Similarly, the producer (e.g., the keyboard driver) deposits characters using and incrementing the front pointer. When it reaches the end of the current array, it allocates a new one and adds it to the front of the queue.

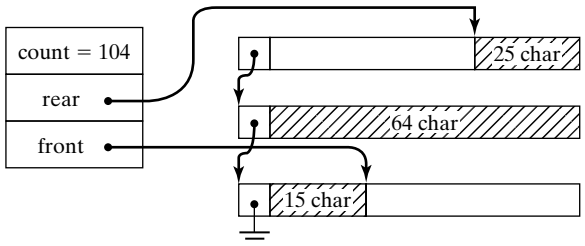

FIGURE 11-15. C-lists in UNIX.

Buffer Cache

FIFO buffers are most useful for stream-oriented devices (e.g., terminals and printers), where the data must be consumed sequentially in the order in which it is produced. For direct-access devices (e.g., disks), data items are accessed in a random order (determined by the program). Furthermore, the same data items may be accessed multiple times. Thus,

the main role of the buffer is to serve as the cache of the recently accessed items to avoid generating them repeatedly.

A buffer cache is generally implemented as a pool of buffers. Each is identified by the disk block it is currently holding. The buffer cache implementation must satisfy the following two requirements. First, given a block number, it must be able to quickly check that this block currently exists in the cache and retrieve its contents. Second, it must facilitate the reuse of buffers that are no longer needed.

To implement the fist requirement, the buffers are generally accessed through a **hash table**, where each entry links together blocks identified by the same hash value. To satisfy the second requirement, the individual buffers are linked together according to a policy, typically least recently used (LRU), so that buffers that are least likely to be accessed again are reused first. This is analogous to the queue of main memory frames used with the LRU page-replacement policy (Section 8.3.1). Additional lists may be implemented to further improve the management or effectiveness of the buffer cache.

CASE STUDIES: DISK BLOCK CACHES

1. *UNIX.* BSD UNIX provides a disk block cache as part of the block-oriented device software layer (Fig. 11-1). This consists of between 100 and 1000 individual buffers. As a result, a large percentage (typically 85% or more) of all I/O requests by the file system and applications can be satisfied by accessing data in main memory rather than the disk. The virtual memory system uses a lower-level interface, which bypasses the buffer cache, but implements its own buffering scheme tailored specifically to its needs.

Each buffer in the buffer cache is represented by a header, which points to the actual variable-size buffer area and records the buffer size, the descriptor of the file to which the data belongs, the offset within the file, and other information. It also contains two pointers to other buffers. The first implement the hash lists, and the second implements four other possible lists on which a buffer may reside.

Figure 11-16 illustrates the concept. It shows an array of hash values, h_i, each pointing at a linked list of buffers whose identifier hashes to the same value h_i. For example, blocks $b71$ through $b75$ are all assumed to hash to the same value $h7$. This allows the system to find a buffer quickly, given its block number. In addition, each buffer is on one of four possible lists. The *locked* list (buffers $b11$ and $b51$ in the figure) contains buffers that cannot be flushed to disk; these are used internally by the file system. The *LRU* list (buffers $b21$, $b71$, and $b72$ in the figure) maintains the buffers sorted according to the LRU policy. A buffer at the end of the list (e.g., $b72$) is reused first, but only when the *Age* list is empty. This links together buffers that have been released, e.g., when a file is deleted, and are not expected to be accessed again. It also contains buffers that have been read in anticipation of future accesses (read-ahead of sequential files). The *empty* list contains buffers whose memory area is currently of size zero. These are only place holders for buffers—they cannot be used again until another buffer shrinks and generates free memory space.

CASE STUDIES: DISK BLOCK CACHES (*continued*)

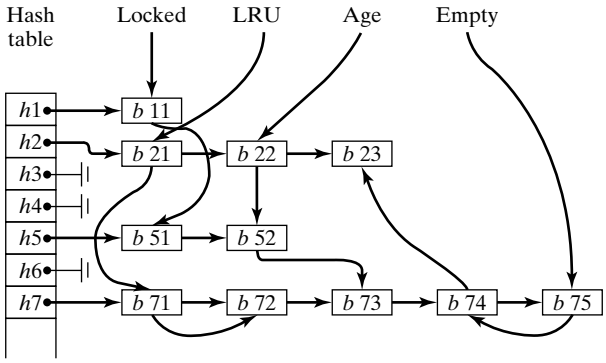

FIGURE 11-16. Buffer cache in UNIX.

2. *Linux.* Linux also uses hashing to find cached blocks. In addition, it segregates all buffers on three types of lists:

(a) Free buffers are kept on seven different lists, according to the buffer size; this starts with 512 bytes and doubles each time.

(b) Buffers containing valid blocks are kept on three different lists: *locked*, *clean* (i.e., read-only), and *dirty* (i.e., already modified); the distinction between clean and dirty avoids unnecessary disk writes when a buffer is evicted from the cache, which is governed by a time-stamp and a number of other system parameters.

(c) Temporary buffers used by the paging system when serving a page fault. These are freed as soon as the operation completes.

11.5.2 Error Handling

Most storage and communication devices contain mechanical moving parts, making them highly prone to error. Furthermore, devices are made by a variety of different manufacturers and may be replaced by newer models several times during the system's lifetime. It is a major challenge to assure a failure-free interaction with such devices.

Errors can be subdivided along different criteria. **Transient** errors or faults are those that occur only under very specific circumstances that are difficult to replicate, such as fluctuations in voltage or a particular timing of events. **Persistent** errors are those that occur consistently and predictably whenever the same program is repeated. A broken wire, a scratched storage disk surface, or a division by zero are examples of persistent faults.

We also can subdivide errors into **software** and **hardware** errors. Persistent software errors can be removed only by correcting and reinstalling the offending programs. Transient software errors also should be corrected in that way, but they are generally very difficult to trace and analyze, since they cannot be easily replicated. If these errors are not too frequent or disruptive, they are generally accepted as a necessary "fact of

life." Some transient software faults are even expected to occur and special mechanisms are put into place to address them. A typical example is the transmission of data through a network. Unpredictable network delays, inadequate resources (e.g., buffers), or other circumstances beyond anyone's control frequently result in data packages being lost or damaged. It is then up to the communication protocols to perform the necessary **retransmissions** or attempt to correct some of the corrupt data using **error-correcting codes**. Such codes require redundant data bits that in some way reflect the data contents to be transmitted with the actual data. The number of redundant bits determine how many corrupt data bits can be detected or corrected.

Transient hardware errors, such as the failure of a disk drive to correctly position the read/write head over a specified cylinder, or a read failure due to a dust particle on a disk block, also can be corrected by **retrying** the operation. Only when the fault persists must it be reported to the higher-level processes. But there are certain types of persistent hardware errors that can effectively be handled by the OS, without involving the user or other high-level processes. The most common such errors are partial **storage media failures**, notably, the destruction of a block on a magnetic disk. These failures are quite frequent, since the magnetic coating on the disk surface is easily damaged. Replacing the entire disk to handle the failure of a single block is not an acceptable solution. Rather, the system must be able to detect failed blocks, generally called **bad blocks**, recover from the failure, and continue working with disks containing such blocks. Several possible solutions, sometimes in combination with one another, address different aspects of the media failure problem.

Bad Block Detection and Handling

Before a disk can be used to hold data, it must be formatted. This process divides the available disk space into blocks (sectors) of a specific size and, with each block, provides some amount of redundant information for the detection and correction of errors. A single parity bit (computed, for example, as the exclusive OR of all bits in the block) would allow the detection of a single failed bit. More generally, a multibit Hamming code attached to each block allows the detection and correction of more than one bit; the numbers depend on the size of code used.

The parity bit or error-correcting code is computed and written together with each disk block write, and it is checked during each disk block read. When an error is detected, the read is retried several times to handle any transient read failures. If the error persists, the block is marked as permanently damaged and all future accesses to it are avoided. This must be done transparently to the software to avoid any changes to the I/O system or application programs.

From the driver point of view, the disk is a continuously numbered sequence of n logical blocks, say $b[0], \ldots, b[n-1]$. When a block $b[k]$ becomes damaged, simply marking it as unavailable would create a gap in the sequence and greatly complicate the I/O software. A more efficient solution is to let the disk controller handle all the mappings of logical to physical blocks internally and transparently to the I/O software. Specifically, when a block becomes inaccessible, the controller changes the mapping so that the software can continue using the same sequence of logical blocks without any gaps.

To make such remapping transparent, some blocks must be reserved initially as **spares**; otherwise, the loss of a physical block would reduce the number of logical blocks. Assume that the disk consists of a total of np physical blocks, where $np > n$.

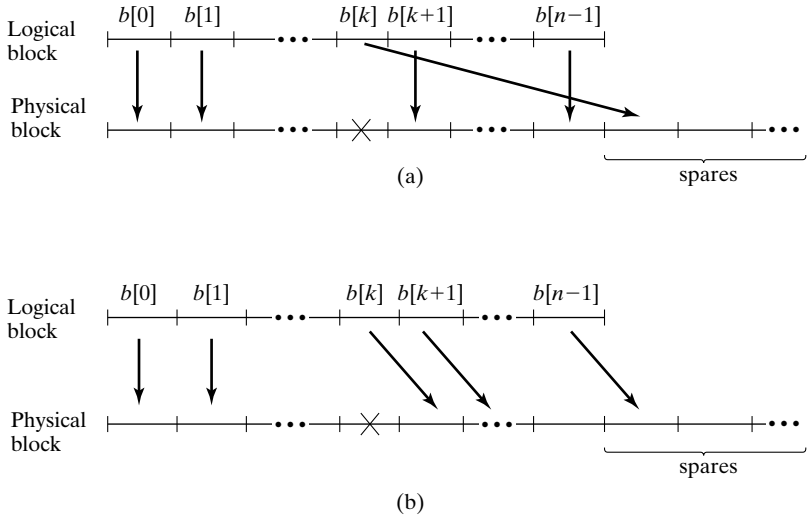

FIGURE 11-17. Handling of bad blocks: (a) remapping; and (b) shifting.

When the physical block holding logical $b[k]$ becomes damaged, the controller can remap the blocks either by allocating a spare to the block $b[k]$ or by shifting all blocks $b[k]$ through $b[n-1]$ to the right by one block. Figure 11-17 shows the two schemes graphically.

The advantage of the first solution (Fig. 11-17a) is that only one block must be remapped, but it can lead to performance degradation if the spare block is on a different cylinder than the original failed block. Let us assume that blocks $b[k-1]$, $b[k]$, and $b[k+1]$ originally all reside on the same cylinder. To read the sequence of these three blocks requires a single seek operation. When $b[k]$ is remapped to a different cylinder, three seek operations are required to read the same sequence of blocks.

One way to minimize this problem is to allocate a number of spare blocks on every cylinder and attempt to always allocate spares from the same cylinder as the failed block. The method of Figure 11-17b shifts all blocks following the faulty one to the right. This requires more work to repair the disk, but it eliminates the problem of allocation noncontiguity.

Stable Storage

When a disk block become defective, some of the information it contains is unreadable. If error-correcting codes are used and the damage affects only as many bits as can be corrected, the information can be recovered and stored in a new block. But when the damage is more extensive, the information is simply lost. Another source of potential data loss is a system crash. If a crash occurs when a disk write operation is in progress, there is no guarantee that the operation was carried out successfully and completely. A partially updated block is generally incorrect and considered lost.

Loss of data is always undesirable but the consequences depend on the applications that use this data. In many instances, the application can be rerun and the lost data reproduced. In other cases, lost data may be recovered from backup files. However,

many applications, such as continuously operating database systems or various time-critical systems logs, require data to be available, correct, and consistent at all times.

There is no system that can eliminate potential data loss with an absolute guarantee; no matter how many levels of backup a system chooses to provide, there is always a scenario that will defeat all safeguards. But we can reduce the risk to a level that is acceptable in any practical application. One way to achieve an extremely high level of data safety is with **stable storage**. This method uses multiple independent disks to store redundant copies of the critical data and enforces strict rules in accessing them.

Consider a stable storage with two disks, A and B. Under failure-free operations, the two disks contain exact replicas of each other. Assuming that both disks do not fail at the same time, one of them always contains the correct data, provided that the following protocols are obeyed by every read and write operation, and during recovery:

- **Write.** Write the block to disk A. If successful, write the same block to disk B. If either of the two writes fails, go to the recovery protocol.
- **Read.** Read the block from disk A and B. If different, go to the recovery protocol.
- **Recovery.** We must differentiate between two types of faults: 1) a spontaneous media failure, where one or more blocks (possibly the entire disk) become inaccessible; and 2) a system crash during one of the read or write operations:

  1. When a read or write operation fails, the block on one of the disks is defective. The data from the corresponding block on the other, still correct disk, are read and restored on a new block on both disks. The failed read/write operations then can be repeated.

     Figure 11-18a illustrates the situation where a block on disk B becomes damaged. The data, X, from the corresponding block on disk A are used to recover from this media failure.
  2. When the system crashes during a read operation, the read can simply be repeated after the system recovers. When the system crashes while writing to disk A, disk B's data are correct and can be used to restore disk A. Conversely,

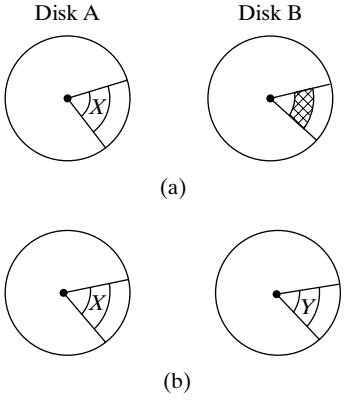

FIGURE 11-18. Stable storage: (a) bad block; and (b) block mismatch after system crash.

when the system crashed while writing disk B, disk A's data are correct and used to restore disk B.

Figure 11-18b shows that two corresponding blocks on the disks A and B contain different data, X and Y, respectively. This discrepancy is due to a system crash that occurred during the write operation. If this occurred before the write to disk A completed successfully, the data Y is correct and replaces the data X. If the crash occurred after disk A completed the write, X is correct and is used to replace Y. After recovery, both disks are again exact replicas of each other. Furthermore, the system knows whether the repaired block contains the old data that resided in the block prior to the write operation or the new data placed there by the write operation.

The assumption about the two disks failing independently is crucial for the stable storage concept to work. The probability of both disks failing at the same time is the product of the probabilities of the failures of each disk, which for all practical purposes is negligible. The addition of more disk replicas decreases the probability further, but it can never be made zero, especially when considering the possibility of catastrophic failures resulting from natural disasters (fires, floods, earthquakes), terrorism, or war.

Redundant Array of Independent Disks

The acronym RAID stands for **Redundant Array of Independent Disks**. Its purpose is increased **fault tolerance**, increased **performance**, or both. Fault tolerance is increased by maintaining redundant data on different disks. Performance, as measured by increased rate of data transfer, is increased by distributing the data across multiple disks. Because I/O can proceed in parallel over each disk, the total achievable data transfer rate of the RAID is the sum of the data transfer rates of the individual disks.

There are different types of RAIDs, depending on the level of data redundancy employed. The simplest form, shown in Figure 11-19a, is an extension of the stable storage concept. It provides an exact replica of each of the primary disks, such that for each block, b_i, there is a copy, b_i', on another disk. Such a RAID is highly fault-tolerant but also very wasteful, as it duplicates the amount of storage needed.

Figure 11-19b shows a RAID organization where only a limited amount of information, derived from the primary data, is kept on additional disks. This information,

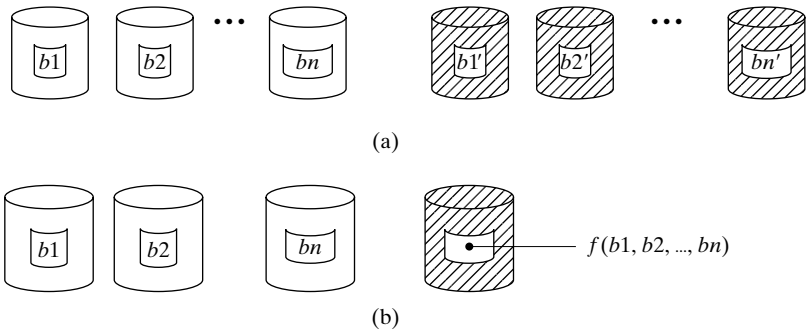

FIGURE 11-19. RAID (a) fully replicated data; (b) derived recovery information.

denoted as $f(b_1, b_2, \ldots, b_n)$, could be simple parity computed across the corresponding bits or blocks of the primary disk, or it could be error-correcting code, capable of detecting and correcting the failure of a number of bits on the primary disks.

Within this basic organization, there are several different types of RAIDs. The basic trade offs are between the amount of redundant information (which increases the number of disks) and the achieved level of fault tolerance (i.e., number of bits that can be corrected). Redundant information can be segregated on separate dedicated disks or interspersed with the primary data. The placement plays a major role in performance; it determines the efficiency (access time and bandwidth) of data access during both normal operation and recovery from failure.

11.5.3 Disk Scheduling

The time to access a block on a rotating magnetic disks consists of three components. The **seek time** is the time to position the read/write head of the disk over the cylinder containing the desired block. The **rotational latency** is the time to wait for the desired block to pass under the read/write head. The **transfer time** is the time to copy the block to or from the disk.

The seek time requires mechanical movement of the read/write head and is the longest. It depends on the physical distance traveled, expressed by the number of cylinders that must be traversed. Seeking to an adjacent cylinder typically takes a few milliseconds; the average seek time to a random cylinder for a hard disk drive is approximately 10 ms and as much as 5 to 10 times that for a floppy disk drive. The rotational delay depends on the rotational speed of the disk, which ranges roughly between 5 to 15 ms per revolution for a hard disk. The rotational delay is, on average, half of that time. The actual data transfer time is the least costly, typically taking less than 0.1 ms for a sector on a hard disk, and less than 1 ms for a sector on a floppy disk. (See Figure 11-5 for typical ranges of disk performance characteristics.)

The above numbers show clearly that both the seek time and the rotational delay should be minimized to achieve good disk performance. This requires two levels of optimization. First, blocks that are likely to be accessed sequentially or as groups must be stored closely together, ideally, on the same cylinder. This is a matter of file allocation, which we have already discussed in Chapter 10. Keeping blocks belonging to the same file clustered within a small number of cylinders decreases the number of seek operations and their travel distance for any given process.

Unfortunately, requests to a disk may come from multiple processes concurrently. The interleaving of these requests would negate much of the locality of access. Thus, additional optimization techniques must be employed at runtime to continuously order the sequences of incoming requests; the ordering maintains the locality of reference, improving the disk performance.

Since both the seek time and the rotational delay are significant, we must order both, requests to different cylinders and requests to individual blocks within each cylinder. Let us first consider the ordering of requests for blocks within a single track. Since the disk always spins in one direction, the solution is straightforward. Assuming that the read/write head passes over the blocks on a track in ascending order, then, given a list of blocks to be accessed on the track, the list is also sorted in ascending order. In that way, all blocks on the list can be accessed during the same revolution of the disk.

With a multisurface disk, the same technique is applied sequentially to all tracks within the same cylinder. The list of blocks to be accessed on each track is sorted and accessed during a single revolution. Thus, it takes up to *s* revolutions to access all blocks within a given cylinder, where *s* is the number of surfaces (i.e., the number of tracks per cylinder.) The read/write head does not move during this time.

The ordering of accesses to different cylinders is a much more difficult problem because the read/write head moves back and forth across different cylinders. Thus, at any point in time, we must decide in which direction the read/write head should move next. The issue is complicated by the fact that requests to access different blocks arrive dynamically as an open-ended stream. They are placed into a queue where they await their turn. Whenever the disk completes the current operation, a process, called the **disk head scheduler**, must examine the current contents of the queue and decide which request to service next.

The scheduler can follow a number of different strategies in making its decision. The three most important factors to consider are: 1) the *overall performance* of the disk, which is reflected by the sustained data rate of the disk; 2) the *fairness* in treating processes accessing the disk concurrently; and 3) the *cost* of executing the scheduling algorithm.

To illustrate the different algorithms, assume the following scenario. The read/write head starts at cylinder 0. The scheduler moves it to satisfy a first request at cylinder 5, where it services all requests for blocks on that cylinder. After completing the latter, the queue contains a sequence of three new requests, to cylinders 12, 4, and 7, in that order. The following three algorithms show in which order these cylinders can be visited and the trade offs between the choices:

First-In/First-Out

The simplest approach to disk scheduling is to service the requests in the order of arrival in the queue, i.e., FIFO. For our example, the scheduler would move the read/write head from cylinder 5 to cylinder 12; then to cylinder 4, and finally to 7. Figure 11-20a tracks the movements of the read/write head. The total number of cylinders traversed is 23.

While very simple to implement, the main drawback of the FIFO algorithms is that it does not attempt to optimize the seek time in any way. It is suitable in systems where the disk is not a performance bottleneck. In such situations, the queue would contain only one or a very small number of requests most of the time. FIFO also may be adequate in single-programmed systems, where the sequences of block accesses already display a significant level of locality resulting from file-block clustering and sequential access.

Shortest Seek Time First

Multiprogrammed and time-shared systems generally place a heavy burden on the disk, and their access patterns show little locality because of the interleaving of requests from different processes. The **shortest seek time first** (SSTF) algorithms addresses the problem by always choosing the closest cylinder to access next, minimizing overall seek time. In the above example, the scheduler would reorder the three requests in the queue to access cylinder 4 first, since this is the closest one to the current cylinder 5. Next it would move to cylinder 7 and finally to 12. Figure 11-20b tracks the movement of the read/write head, which traverses a total of only 14 cylinders.

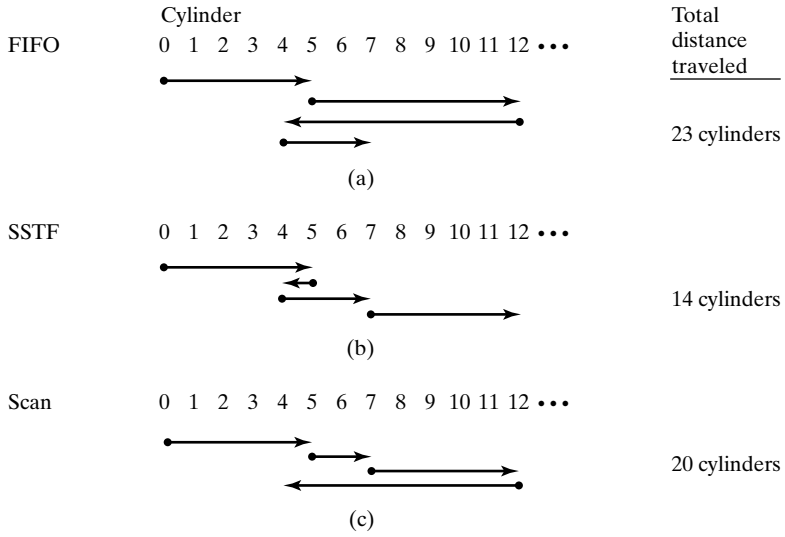

FIGURE 11-20. Disk scheduling algorithms: (a) FIFO; (b) SSTF; and (c) scan.

The SSTF algorithm minimizes the total seek time and maximizes the total bandwidth of the disk. Its main drawback is the unpredictability with which it services individual requests. Consider the following situation. When accessing cylinder 4, new requests for cylinders 3, 6, 2 arrive in the queue. The SSTF will service all of these before it gets to cylinder 12. In principle, an unbounded sequence of requests for cylinders in the general vicinity of cylinder 5 would delay the servicing of cylinder 12 indefinitely. Such unpredictability and possible starvation of outlying requests are not acceptable for a general time-sharing system that must guarantee some level of acceptable response for all users.

Elevator or Scan

A good compromise between the greedy nature of the SSTF and the need to maintain fairness in access is the elevator or scan algorithm presented in Section 3.3.3. Recall that the basic strategy is to continue to move the head in the same direction as far as possible and then change to the opposite direction. We can use the same *elevator* monitor for the disk head scheduler. The main difference is that with a real elevator, requests come from two distinct sources: call buttons at each floor and destination buttons inside the elevator cage. With a disk head scheduler, there is only a single stream of requests for different cylinders (destinations) issued by different processes. Thus, the I/O system translates each request for a particular cylinder into a call *elevator.request(dest)* where *dest* is the cylinder number. After it has read the corresponding blocks on that cylinder, it issues the call *elevator.release()*, which enables the next request to proceed.

Figure 11-20c tracks the movement of the read/write head under the scan algorithm. When it is on cylinder 5, its current direction is up and it does not reverse to service the closest cylinder 4, as in the case of SSTF. Instead, it continues in the same direction to service cylinders 7 and 12. Cylinder 4 is served on the down-sweep. The total number of

cylinders traversed in this example is 20, which lies between the two previous extremes. The increase over SSTF is the price one pays for the greatly improved fairness in servicing individual requests—no request can be starved indefinitely.

There are several variations of the basic scan algorithm to further improve its behavior. The most important of these, **circular scan**, services requests only in one direction, say from lowest to highest. When it reaches the highest cylinder, the read/write head simply moves back to cylinder 0 and starts the next scan. This further equalizes the average access times to blocks; with the simple scan algorithms, tracks closer to the middle of the disk are serviced sooner on average than cylinders at the fringes.

11.5.4 Device Sharing

Some devices, such as disks, can be used concurrently by multiple processes. It is up to the scheduler to order the requests and present them to the driver, as discussed earlier. However, many devices, including keyboards, terminals, printers, or tape drives are only serially reusable. That means, they can be used by only one process at a time; otherwise, meaningless interleaving of data can occur.

To provide serial-reusability, most I/O systems have interface commands that permit processes to **request** and **release** individual devices. As already discussed in Section 11.2.1, some systems (notably UNIX/Linux) treat devices as streams of data similar to files. Thus, the same ***open*** and ***close*** commands used to request and release access to files are used to request and release devices. When a device is currently in use, the *open* command will block the requesting process, or it may return with a message indicating the state of the device. The use of *open/close* to control the sharing of devices provides for a uniform interface to access the file system and the I/O system.

Spooling, introduced briefly in Chapter 1, is frequently implemented for noninteractive output devices, such as printers. The output data of a process is not sent immediately to the printer as it is being generated but is written into a file instead. The file serves as a **virtual printer**. With this organization, multiple processes can output data concurrently without reserving the actual printer. The output files are kept in a special spooling directory. A dedicated server process then accesses these files sequentially and sends them to the printer one at a time. File transfers, email, and other communication through networks are handled in a similar way to optimize the use of underlying networks.

CONCEPTS, TERMS, AND ABBREVIATIONS

The following concepts have been introduced in this chapter. Test yourself by defining and discussing each keyword or phrase.

Bad block	Input-output (I/O)
Block-oriented device	Joystick
Buffer swapping	Laser printer
Buffering	LCD/LED printer
CD-ROM	Memory-mapped I/O
Circular buffer	Network communication interface
Circular scan	Network interface card (NIC)
Communications device	Optical disk

Connection-based and connection-less protocols
Consumer process
Cathode ray terminal (CRT)
Cycle stealing
Digital audio tape (DAT)
Device controller
Device driver
Device sharing
Direct memory access (DMA)
Disk cylinder
Disk scheduling
Disk sector
Disk track
Digital linear tape (DLT)
Dot-matrix printer
Ethernet
FIFO disk scheduling
Flat-panel monitor
Floppy disk
Hard disk
Hierarchical I/O systems model
Impact printer
Ink-jet printer
Interrupt-driven I/O
I/O device
I/O errors
Packet
Pixel
Polling
Producer process
Programmed I/O
Read/write head
Redundant array of independent disks (RAIDS)
Scan algorithm
Scanner
Seek time
Socket
Spooling
Shortest seek time first (SSTF)
Stable storage
Storage device
Stream-oriented device
Thermal printer
Track skew
Trackball
Transfer time
Transmission Control Protocol/Internet Protocol (TCP/IP)
Universal Datagram Protocol/Internet Protocol (UDP/IP)
Video RAM

EXERCISES

1. A fast laser printer produces up to 20 pages per minute, where a page consists of 4000 characters. The system uses interrupt-driven I/O, where processing each interrupt takes 50 μsec. How much overhead will the CPU experience if the output is sent to the printer one character at a time? Would it make sense to use polling instead of interrupts?

2. Consider a 56-Kbps modem. Transmitting each character generates an interrupt, which takes 50 μsec to service. Assuming each character requires the transmission of 10 bits, determine what fraction of the CPU will be spent on servicing the modem.

3. Assume a mouse generates an interrupt whenever its position changes by 0.1 mm. It reports each such change by placing a half-word (2 bytes) into a buffer accessible by the CPU. At what rate must the CPU retrieve the data if the mouse moves at a speed of 30 cm/second?

4. How much longer does it take to completely fill the screen of a color graphics monitor with 800×600 pixels and 256 colors than the screen of a character-oriented, black-and-white monitor with 25 lines of 80 characters? Repeat the calculation for a graphics monitor with double the resolution, i.e., 1600×1200 pixels.

5. Consider a two-sided floppy with 80 tracks per surface, 18 sectors per track, and 512 bytes per sector. Its average seek time between adjacent tracks is 2 ms, and it

rotates at 500 rpm. Determine the following:
(a) The total storage capacity of this disk.
(b) The number of seek operations that must be performed to sequentially read the entire content of the disk.
(c) The peak data transfer rate.
(d) The optimal amount of track skew.

6. Consider a single-sided floppy with 80 tracks, 18 sectors per track, 512 bytes per sector, an average seek time of 30 ms, a seek time between adjacent tracks of 3 ms, and a rotational speed of 360 rpm. A sequential linked file (Figure 10-12b) consisting of 12 blocks is stored on the disk in one of the following ways:
(a) The 12 blocks are spread randomly over all tracks.
(b) The file is clustered, such that the 12 blocks are spread randomly over only 2 neighboring tracks.
(c) The 12 blocks are stored in consecutive sectors on the same track.
How long will it take to read the entire file in the three cases?

7. Consider a disk consisting of c cylinders, t tracks within each cylinder (i.e., t surfaces), and s sectors (blocks) within each track. Assume all sectors of a disk are numbered sequentially from 0 to $n-1$ starting with cylinder 0, track 0, sector 0. Derive the formulas to compute the cylinder number, track number, and sector number corresponding to a given sequential block number k, $0 \leq k \leq n-1$.

8. Two identical files, $f1$ and $f2$, of 100 KB each are stored on two different disks. The file blocks are spread randomly over the disk sectors. Both disks have an average seek time of 10 ms and a rotational delay of 5 ms. They differ in the organization of their tracks. Each track of disk 1 consists of 32 sectors of 1 KB each, and each track of disk 2 consists of eight sectors of 4 KB each. How long will it take to sequentially read the two files?

9. Consider a disk holding files with an average file length of 5 KB. Each file is allocated contiguously on adjacent sectors. To better utilize the disk space, it must be periodically compacted, which involves reading each file sequentially and writing it back to a new location. Assuming an average seek time of 50 ms, 100 sectors per track, and a rotational speed of 5000 rpm, estimate how long it will take to move 1000 files.

10. Consider a program that outputs a series of records and make the following assumptions:
- It takes two units of CPU time to perform an I/O instruction; all other instructions take 0 units of time.
- The device starts at the end of the I/O instruction, and it remains busy for 6 time units.
- When polling is used, the CPU polls the device every 4 time units. The first poll occurs immediately after the I/O instruction. That means, if the I/O instruction ends at time i, the first poll occurs at time $i+1$. Assume that a poll takes 0 units of CPU time.
- When interrupts are used, the interrupt handler routine takes 5 time units
- At time $t = 0$, the device is not busy and the CPU starts executing the program

Draw a timing diagram showing when the CPU and the device are busy during the first 25 time units (a) with polling; and (b) with interrupts.

11. Section 11.4.2 shows the pseudocode for performing input and output using programmed I/O with polling. Consider a process that must read a sequence of n characters and output a sequence of n other characters. Make the following assumptions:
- The two character sequences are independent and thus may be processed concurrently.
- Two separate devices are used: one for input and one for output. The two devices are started using the instructions *write_reg(opcode1)* and *write_reg(opcode2)*,

respectively. Similarly, each device has its own busy flag and data buffer (referred to as 1 and 2).

Write the pseudocode for the process to work as fast as possible.

Hint: Combine the two separate code segments of Section 11.4.2 to achieve maximum overlap between the processor, the input device, and the output device.

12. Consider again the pseudocode of Section 11.4.2 (programmed I/O with polling). Make the following assumptions:

- Each line of the code takes 1 unit of CPU time to execute.
- The *write_reg()* function starts the I/O device. The device becomes busy at the end of this function and remains busy for 10 units of CPU time.

(a) Draw a timing diagram for the input code, showing the sequence of instruction execution and the time the device is busy for the first 30 time units.

(b) Draw a timing diagram for the output code, showing the sequence of instruction execution and the time the device is busy for the first 30 time units.

(c) How long does it take to first input n characters and output n characters?

(d) Draw a timing diagram for the combined I/O code of the previous exercise, showing the sequence of instruction execution and the time the two devices are busy for the first 40 time units.

(e) How long does it take to concurrently input n characters and output n other characters using the program of the previous exercise?

13. Section 11.4.3 shows the pseudocode for performing input and output using programmed I/O with interrupts. Note that the device must remain idle during the memory copying operation, but it need not be idle during the *increment i* and *compute* operations. Assume that the compute statement has the form *compute(i)*, where i indicates the character to be processed. Rewrite the code for both input and output to achieve maximum overlap between the CPU and the device. That means, when the input devices reads character i, the CPU should be computing with character $i - 1$. Similarly, when the output device writes character i, the CPU should be preparing character $i + 1$.

14. Consider a single consumer process that communicates with n independent producers. Each producer uses a separate buffer to present data to the consumer. Assume that the time f to fill a buffer is larger than the time c to copy and process the buffer, i.e., $f = kc$, where $k \geq 1$ is an integer.

(a) Draw a timing diagram (similar to Fig. 11-9) showing the concurrent operation of the consumer and the different producers (each filling one of the n buffers).

(b) How many buffers can be used to keep the processor busy all the time, i.e., what is the optimal value of n?

(c) What is the ideal number of buffers when $k = 1$, i.e., $f = p$? Does it still help to have multiple independent producers?

(d) What is the ideal number of buffers when $k < 1$?

15. A RAID consisting of n disks uses a four-bit error correction code for each 512-byte disk block. The codes are not stored on a separate disk but are spread over the n disks, such that any disk i holds the codes of disk $i + 1$ (modulo n). Determine what fraction of the total disk space is taken up by the error codes.

16. Consider a disk with n cylinders numbered sequentially $0, 1, 2, \ldots, n - 1$. The unit has just finished servicing requests on cylinder 125 and has moved to cylinder 143. The queue of requests to be serviced is as follows:

143, 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

where 143 is the oldest request and 130 is the latest arrival.

(a) In which order will the cylinders be serviced under the following scheduling policies:
- FIFO
- SSTF
- Scan

(b) What will be the distance (number of cylinders) traveled by the read/write head under each policy?

(c) Consider a variant of the scan algorithm, where the read/write head services requests only in one direction, say during its up-sweep. When it reaches the request with the highest cylinder number, it moves down to the request with the lowest cylinder number, without servicing any requests on its down-sweep. The advantage of this strategy, referred to as *circular scan*, is that the disk distributes its service more uniformly over all cylinders. In comparison, the regular scan gives preferential treatment to cylinders closer to the center of the disk. Determine the number of cylinders the read/write head must traverse to service all the requests in the above queue using circular scan.

17. Simplify the code of the *elevator* monitor (Section 3.1.1) to follow the FIFO scheduling algorithm instead of the elevator algorithm.

18. Consider the following implementation of a disk head scheduler. It uses a single-priority wait queue, *q*; processes are placed into this queue based on their distance to the variable *headpos*, the current head position. Construct a scenario illustrating why this does *not* correctly enforce the SSTF algorithm.

```
monitor disk_head_scheduler {
  int headpos=0, busy=0;
  condition q;

request(int dest) {
  if (busy) q.wait(abs(dest - headpos));
  busy = 1;
  headpos = dest;
}

release() {
  busy = 0;
  q.signal;
}
}
```