CHAPTER

10 I/O Device Management

Learning Objectives

After reading this chapter, you should be able to:

- Describe an I/O controller and its interface to the system.
- Describe the inner-workings of a typical I/O controller, namely SCSI HBA.
- Explain how CPU and I/O controller interacts.
- Explain a device driver and how it manages the I/O devices.
- Describe the inner-workings of the two I/O devices—disk and network interface card.

10.1 Introduction

Many varieties of input–output (I/O) devices are used in modern computers. Where their usage is concerned, I/O devices are the most complicated hardware resources. They provide "difficult-to-use" intricate interface(s), and hence, it is not easy for (novices) application developers to handle I/O devices properly. The devices are normally manipulated by special software programs called *device drivers* written by experienced system program developers. In this chapter, we examine how operating systems manage I/O devices. We also study the role of the I/O subsystem in managing a variety of I/O devices via device drivers. The subsystem schedules I/O requests to devices, allocates devices to processes, caches data from the I/O devices, etc. In this chapter, we will also study two widely used I/O devices, namely the disk and the network interface card.

≫ The only way a processor can interact with the outside world is through I/O devices. As I/O devices come in a variety of forms, the subsystem which manages these devices comprises the messiest part of operating systems. The I/O subsystem and device drivers occupy a major part of the operating systems.

10.2 I/O Devices

Many varieties of I/O devices are used in modern computers. Examples of I/O devices include disks, tapes, CDs, printers, network interface cards, the keyboard, the monitor, etc. Some are purely input devices, some purely

Scanned by CamScanner

output devices, and others truly input-output devices. For example, a keyboard is an input-only device, a display monitor is an output-only device, and a disk is an input-output device. However, in this book they all are referred to simply as I/O devices. Some devices transfer transient data from one hardware component to another and others store data persistently. These latter devices can retain data across power disconnections, and are called *storage devices*. In contrast, the other I/O devices are called communication devices—those that the CPU uses to communicate with the outside world.

» Many authors use I/O controller and I/O device to mean the same, but we consider them as distinct from each other. An I/O device is a piece of hardware which does the actual work, and an I/O controller comprises electronic components that operate the devices. Users always think in terms of I/O devices, not of I/O controllers.

As stated earlier, I/O devices are the most complicated hardware resources in a computer. The devices have different physical organizations with various physical characteristics such as the recording medium, the storage capacity, interface operations, and the access speed. Some devices transfer one byte of data at a time, and some a block (fixed-sized multiple bytes) of data at a time. Some are sequential devices and some are random access devices. Interface operations vary from one device to another. Each device needs a unique way to handle it. In this section, we discuss how I/O devices are connected to the host system, and how the latter operates those devices.

10.2.1 I/O Controllers

All I/O devices are connected to the host computer system through an I/O controller (see Fig. 2.10 on page 61). Some I/O devices come with built-in dedicated I/O controllers. A single I/O controller may, however, control many similar devices. The I/O controllers reside on the host system, and they are called *peripheral devices* in contrast to the I/O devices proper. The CPU always accesses an I/O device through the device's I/O controller. An I/O device is typically connected to an I/O controller through a set of wires, collectively called an *I/O bus*.

» Some authors call the bus that connects I/O controllers to the processor the I/O bus. You have been warned for the confusion! Wireless devices are not physically connected to the computer system. They communicate with their I/O controllers using "over-the-air" tranmitted signals.

An I/O controller is a peripheral device that enables the main processor to transfer data between the host system and the I/O devices. The I/O controllers are special purpose processors and are autonomous in the sense that they carry out operations on I/O devices while the main CPU continues to execute programs. The CPU interacts with an I/O controller through a set of controller interface registers. These registers are called *I/O ports* or *operating registers* of the controller. The CPU executes I/O instructions in addressing these ports individually. The ports constitute the I/O address space of the controller. The I/O address spaces of all I/O controllers together constitute the *I/O address space* of the computer system.

» In Linux systems, the *proc/ioports* gives information about which I/O ports are assigned to which I/O devices.

The CPU controls the activities of an I/O controller by writing into and reading from the controller I/O ports. The processor architecture supports special machine instructions to read and write controller I/O ports. For the purpose of communications between the CPU and an I/O controller, the controller's I/O ports are broadly classified into four groups: (1) command, (2) status, (3) input, and (4) output, (see Fig. 1.6 on page 16). The command and status registers are used to start and stop all devices connected to the controller, to initialize them, and to diagnose any problems encountered with them. To start an I/O operation, the CPU writes appropriate directives in

Scanned by CamScanner

command ports and input data to input ports. The controller, in turn, executes the command, and, when the command execution is complete, writes back the command execution status to the status ports and output data to the output ports. The CPU then reads the status and output ports respectively to find the command completion status and the output the command has produced. Other operating registers such as the configuration registers are used to configure a controller at the time of its initialization. Sometimes, a single operating register is used for multiple purposes. Reading and writing of I/O ports are done by executing special IN/OUT instructions.

The synchronization between the CPU and an I/O controller is performed by the interrupt processing or by busy wait handshaking. An I/O controller may or may not have an embedded interrupt circuit. If it does not, the CPU will constantly (or intermittently) monitor a command execution by reading the controller status register. If the controller has an embedded interrupt circuit, the CPU may go on to some other activity after initiating an I/O command. On the completion of command processing, the controller interrupts the CPU. The CPU, on receiving the interrupt, reads the command execution status information from the status ports. The former is called *programmed I/O*, and the latter is called *interrupt-driven I/O*.

≫ Some devices support mapping of I/O addresses into memory addresses. For these devices, the I/O ports can be accessed by executing ordinary memory read and write instructions against the corresponding memory addresses. The system with this addressing scheme is called the *memory-mapped I/O*. If the I/O ports use a separate address space, then they are called *port-mapped I/O*.

10.2.2 DMA Mode Data Transfer

The I/O controllers help in the exchange of data between the CPU and the I/O devices proper. The controllers have to obtain the data from or to somewhere in the host system. One way is for the CPU to exchange data with the controllers via their input and output ports by executing I/O instructions. This is the direct mode of data transfer by the CPU. However, when a large amount of data is to be transferred between the host system and an I/O controller, the byte-by-byte (or word-by-word) data transfer through the controller input and output ports is extremely inefficient. In such cases, interrupt handling adds substantial overhead in data transfer. The overhead is on account of saving the values of processor registers on an interruption and restoring them on completion of the interrupt service. If the interrupt service time is significantly lower than the overhead, the computation speed slows down considerably. For example, consider an I/O operation from a disk. A typical disk can transfer several megabytes of data per second. If the disk's I/O controller interrupts the CPU on every byte (or every few bytes) of data transfer, the CPU will spend a substantial amount of time transferring that mass of data. Alternatively, if we employ the handshake mode of data transfer, the CPU spends a considerable amount of time reading the device status and output ports.

≫ The DMA set up time is quite high compared to port read/write time. Thus, when a large amount of data is transferred, we use the DMA mode, but when a few bytes are transferred we use the direct mode of data transfer between the CPU and the I/O controller.

We need assistance from the hardware to transfer such large volumes of data without involving the CPU. Modern I/O controllers are fitted with direct memory access (DMA) devices. (There might be separate DMA devices on the host system too, to help other I/O controllers in transferring data between the host system and them. This instance is depicted in Fig. 10.1.) The DMA mode of data transfer is the most efficient. A DMA device assists the I/O controllers transfer data

Scanned by CamScanner

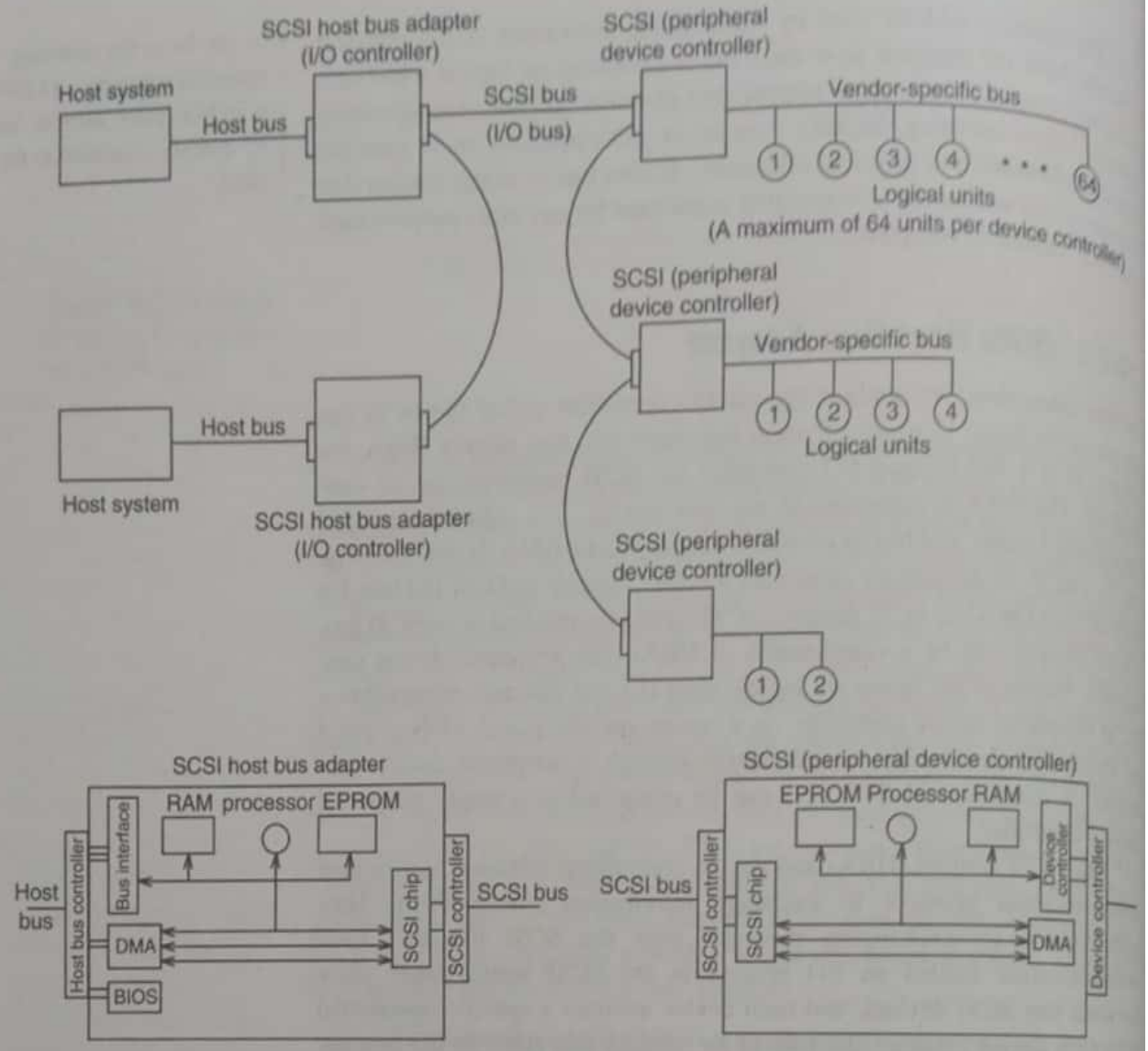


Figure 10.2: *SCSI Host bus adapter and peripheral device controller.*

10.3 Device Drivers

An I/O controller is interfaced with the operating system by a specific piece of software called the *device driver*. On the system software side, the driver is in sole command of manipulating the I/O controller and its connected I/O devices (see Fig. 10.3). The driver implements the software interface that enables the operating system to access data from I/O devices by executing the driver program.

Device drivers know the special characteristics of the I/O controllers and I/O devices they handle, and take advantages of these characteristics. Each driver is customized to a specific I/O controller. The drivers insulate the operating system from a wide spectrum of physical complexities of the I/O devices, and implement a uniform interface for the operating system. Thereby drivers help the operating system handle all devices in a uniform "device-

≫ The KDIM is a generic interface. Any software module that implements the interface is considered a device driver even if it does not manage a physical device.

Scanned by CamScanner

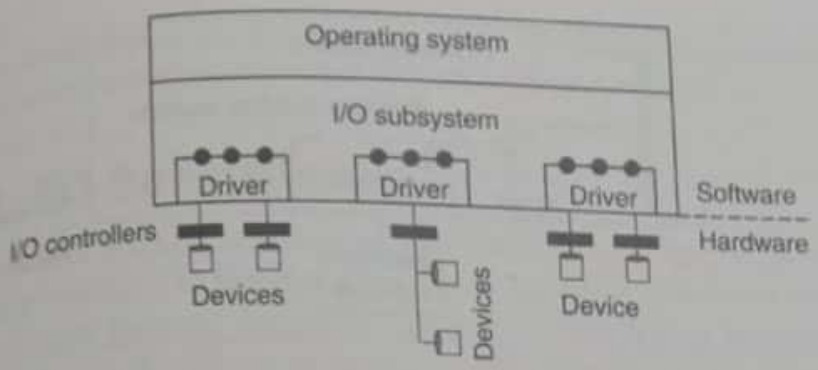


Figure 10.3: *Device driver interface to I/O devices.*

independent" manner. They hide the differences among the I/O controllers from the operating system.

Modern operating systems define their own *kernel-device interface model* (KDIM, for short). The KDIM defines the specifications for the interface between the operating system and all device drivers. The KDIM helps keep the operating system free from intricate device details. Device manipulations are directly done by the device drivers, and not by other parts of the operating system. A device driver implements the KDIM interface specifications on the top of the hardware devices it manages. The driver transforms the "difficult-to-use" device hardware interface into the "easier-to-use" KDIM interface that the operating system can use with relative case.

≫ Each operating system follows its own KDIM, and its model may differ considerably from those followed by other operating systems.

One can visualize a device driver as an abstract object or a monitor accessed by a predefined set of operations. Device-driver software consists of a set of private data structures, a set of device-dependent routines, and those device-independent routines that are specified in the KDIM. When the driver-routines are executed, the driver is said to be controlling activities of the I/O controller and its connected devices.

≫ Note that device drivers reside in the kernel space. Drivers should be written with extreme care. If they go wrong, they can seriously damage the system. A rogue driver may hang the system, possibly corrupting the contents of storage devices. A driver should be tested well before integrating into a user environment.

10.3.1 Driver Initialization

Different types of I/O controllers require different device drivers. There is a device driver for each I/O controller. However, a device driver can control many similar I/O controllers. The operating system interacts with the controllers through their drivers. Consequently, device drivers need to be made known to the operating system. This is done by way of registration at the time of system bootstrapping or when a device driver (as a kernel module) is loaded in the main memory and initialized at runtime. At the time of a driver initialization, the driver registers the KDIM interface functions with the operating system (see Fig. 10.4). The system invokes these functions to access devices managed by the driver. The driver, in turn, may also invoke kernel functions to get specific kernel services such as acquiring and freeing kernel memory.

≫ Drivers are kernel modules. When a driver is loaded, the loader fixes all the unresolved symbols in the driver software using the kernel symbol table.

Some of the functions of a typical device driver are: start and stop device(s), initialize and check for status (on, off, error, etc.), accept operations such as read and write, validate input, manage power, etc.

Scanned by CamScanner

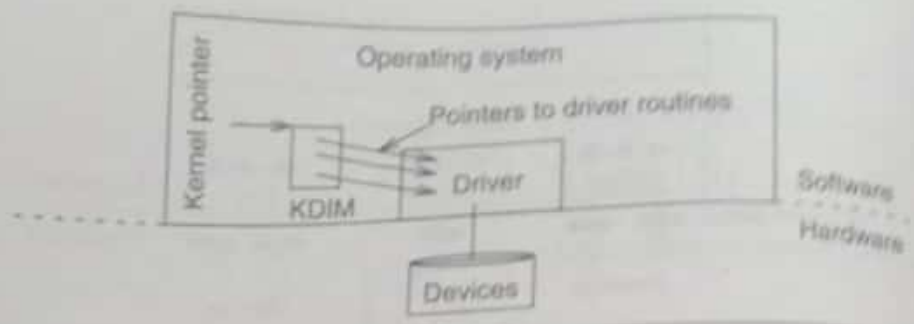


Figure 10.4: *Device driver and kernel interface.*

10.3.2 Software–Hardware Interactions

All a driver does is it assists the host system exchange data and control information with the I/O devices via their I/O controllers. For example, the driver accepts data from the operating system, and instructs the target I/O controller what to do with the data. As mentioned previously, the interactions between the host system and the I/O controller may need to be synchronized. There are essentially two ways to achieve the synchronization: (1) polling and (2) interrupt.

Polling

A device driver instructs an I/O controller on what to do by writing a command and input data into the controller command and input registers, respectively. Then, it continuously (or intermittently) tests the controller's state by reading the controller status register. It does so until the command execution is complete. When the command execution is indeed complete, the driver reads the output data from the controller output register.

≫ Polling is the simplest way to interact with a device. If the I/O device and the controller are fast, it is a reasonable way to interact with the controller.

Polling has some severe consequences if the devices and/or the controllers are slow. So long as the driver polls the controller, the CPU does not execute any other programs until the controller has completed the current I/O request. If, because of defects in the controller, it does not write its status register, the whole system stalls.

Interrupt Service Routines

≫ Devices communicate with the drivers (and the CPU) by interrupts and the drivers (and the CPU) communicate to the devices by writing commands to the device registers.

This synchronization scheme is possible only if the I/O controller hardware has an embedded interrupt circuit. The driver instructs the controller on what to do by writing a command and input data into the controller command and input registers, respectively. Then, the driver returns the control to the operating system, and the CPU goes on to some other activity. When the controller finishes the command, it interrupts the CPU to draw its attention to the controller. The CPU, in turn, suspends its current activity, and executes the interrupt service routine in the driver, where it reads the command completion status and output data from the controller. The operating system maintains a table called the interrupt vector table containing the addresses of the interrupt service routines. The platform hardware and the operating system together

Scanned by CamScanner

connect an interrupt to the appropriate handler. (In Chapter 12 we will study how the operating system dispatches interrupt service routines.)

10.4 I/O Subsystem

Device drivers manage operations on individual devices. They implement a basic set of (uniform) KDIM operations that the operating system can invoke to obtain services from the devices via the drivers. The drivers hide from the operating system the differences among the I/O controllers. All device drivers are managed by the I/O subsystem, a component of the operating system. The objective of the I/O subsystem is to implement a single generic interface for all I/O devices so that the rest of the operating system uses the devices conveniently and efficiently. The subsystem insulates the rest of the operating system from the management of the complex I/O devices. It keeps information pertaining to all device drivers. It also manages allocation and deallocation of exclusive (non-sharable) devices to processes. The I/O subsystem implements a generic driver that drives individual device drivers proper. To improve I/O performance, it also maintains data cache(s) to hold contents from block devices.

≫ Device independency, error handling, and data caching are the main goals of the I/O subsystem. Device independency means the access or the name should not reflect the device, error handling aims to identify and contain errors locally, and caching aims to mask the differences in speed.

10.4.1 Device Types

Many varieties of I/O devices are used in modern computers. They have different physical characteristics. The devices may be characterized on the basis of the following physical attributes:

- **Character or block**: A device may transfer one character/byte or one block (multiple bytes) of data at a time. The byte is the smallest unit of data transfer for a character device, and the a priori known fixed size block is the smallest unit of data transfer for a block device. Different devices can, however, have different block sizes. For example, a keyboard transfers one character at a time, and a typical disk a 1024-byte block at a time.
- **Read-only, write-only, or read-write**: Some devices such as keyboards only supply data to the host system; some others such as printers consume data from the host system; yet some others such as disks both supply and consume data.
- **Speed**: Some devices such as keyboards are slow transferring a few bytes per second, and some such as Ethernet card are fast, transferring millions of bytes per second.
- **Transient or durable**: Some devices such as Ethernet cards store transient data for very short durations. Some devices such as disks store persistent data that are not lost in the event of power disconnections.
- **Sharable or exclusive**: Some devices such as disks are used concurrently by multiple processes. They are not explicitly allocated to any particular process. These devices are allocated to processes for

Scanned by CamScanner

short durations as long as the processes are in the kernel space. Some devices such as the graphics plotter are used only exclusively. They are allocated to processes as long as the processes need them.

- **Access pattern**: A storage device may be viewed as a sequence of data blocks, where each block holds one or multiple bytes.
  - In *sequential access devices*, data blocks are only accessible in a sequential order. The sequential order is defined by device characteristics. It supports minimally three operations: (1) rewind, (2) read, and (3) write. The rewind operation repositions the device to the first data block. A read (respectively, write) execution returns (overwrites) the content of the current data block, and repositions the device to the next data block. Tape devices fall in this category.
  - In *direct access devices*, any data block is directly accessible by providing its ordinal position number in the data block sequence. That is, data blocks can be accessed in an arbitrary order, reasonably fast. Disk devices fall in this category.
  - In a direct access device, if the time required to access a data block is independent of its ordinal position in the data block sequence, then the device is called a *random access device*. Some flash devices fall in this category.

10.4.2 Device Categories

The I/O subsystem classifies all the I/O devices into a couple of (or a few) categories for the convenience of managing them. For example, in UNIX, the I/O subsystem identifies three broad device categories: (1) block devices, (2) character devices, and (3) network devices. This is shown in Fig. 10.5. Block devices include disks and tapes. Their contents are accessed in fixed size blocks. A network device is one that sends and receives packets of data, such as an Ethernet interface card. Character devices include every device that is neither a network device nor a block device. An example of a character device is the keyboard.

≫ Network devices are different from the other two categories in the sense that they receive data from outside the computer system.

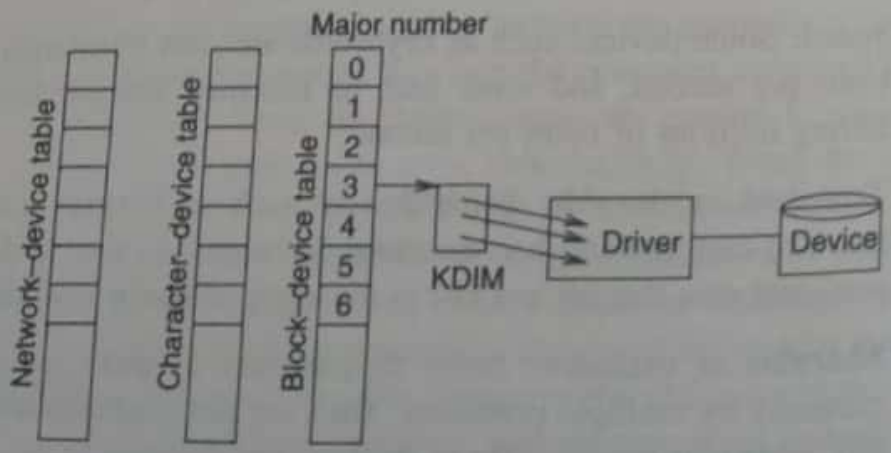


Figure 10.5: *Device driver tables in the I/O subsystem.*

Scanned by CamScanner

The KDIM interfaces are different for the three device categories. The block-device interface specifies routines such as read, write, reposition for accessing disks and other block devices. The character-device interface specifies routines such as get (a character), put (a character) for devices such as the keyboard and other character devices. (Note that block devices can also be accessed through the character-device interface.) All inter-computer communications devices are accessed through the network-device interface. The devices in the former two categories are generally accessed through file system abstraction, and in the last category through the communications system or socket abstraction (see Fig. 3.1 on page 71).

Device drivers are accordingly classified into three different categories. To store information related to device drivers, the I/O subsystem maintains three different tables in the three categories, (see Fig. 10.5). As mentioned previously, devices in each category are further classified into different types. For each category, every type is identified by a distinct integral number called the major number (in UNIX systems). Devices in the same type are uniquely identified by another integral number called the minor number. Internally, a device is identified or addressed by its device category and a device number in the category, where the device number is a pair consisting of a major number and a minor number. The device category and device number act as links between the operating system and the I/O subsystem. The I/O subsystem uses the major number as an index to the appropriate device table to find the device driver information (see Fig. 10.5). In the figure, the block-device table at major number 3 points to a KDIM interface structure. When a device driver is registered with the kernel, a pointer to the driver KDIM structure is stored in an appropriate device table for a given major number. The (KDIM) pointer acts as a link between the I/O subsystem and the driver. To act on a device, the I/O subsystem must be informed about the device category and the major number, using which the subsystem can access the appropriate driver interface routines. The minor number is passed down to the device driver, and is interpreted by the device driver as a partition or line number or individual unit number or whatever.

≫ If you have a Linux machine, you can see that its disks have major number 3, and the /dev/hda1 disk has minor number 1, /dev/hda2 has minor 2, and so on. These minor numbers distinguish one disk (or their partitions) from another.

10.4.3 I/O Subsystem Functionalities

All the I/O subsystem primarily does is to allocate exclusive (non sharable) I/O devices to processes, and schedule I/O requests to all the I/O devices. The system also maintains a cache to hold data from block devices.

Device Status

The I/O subsystem maintains the status of all devices connected to the system. This status information is stored in a device status table (see Fig. 10.6). Each entry in the status table indicates whether the device is idle or busy; if busy, the process that is using the device and the processes that are waiting for the device. The mode indicates whether the device is sharable or exclusive. In the figure, Disk 1 is a sharable device; it is currently busy, being in

≫ Concurrency control is another concern for the I/O subsystem; various synchronization tools (see Chapter 7) are used to make the I/O subsystem highly concurrent.

Scanned by CamScanner

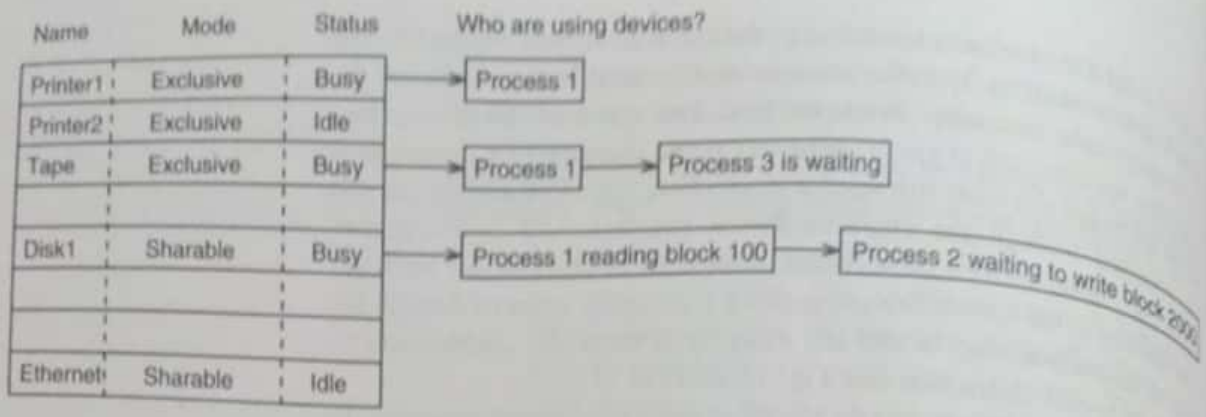


Figure 10.6: *Device status table.*

use by Process 1. This allocation to process 1 is only for a very short duration, and is under the control of the kernel. Process 2 is currently waiting for disk service. In the figure, Tape is an exclusive resource, and it is currently allocated to process 1, and process 3 is waiting for the Tape. The time required by process 1 in using the Tape is not in the control of the kernel. When a device becomes free, the I/O subsystem assigns the device to one of the waiting processes. As noted in Chapter 5, for every device, the kernel has a device allocator that manages the device. The device allocator follows a scheduling algorithm to allocate the device to processes in orderly manner. It may also schedule requests from users to the device.

I/O Request Scheduling

The system performance depends on how effectively the devices are allocated to processes. Different devices normally have different scheduling policies. Application processes issue I/O requests at unpredictable times (through system calls). One way is to carry out these requests on first-come first-serve (FCFS) order. However, FCFS discipline may not always be the best choice in some cases. (We will see an example of non-FCFS disk scheduling in Section 10.5.6.) For every device, the I/O subsystem determines the best order among pending requests, and schedules them in that order. The scheduling order depends on several factors: device configuration, process priority, fairness of scheduling, etc. The scheduling is aimed at improving overall system performance, and reducing the average waiting time on I/O device queues. As shown in Fig. 10.6, for each device, the I/O subsystem maintains one entry in the device status table; the entry has a waiting queue where processes wait until the device is allocated to them.

≫ As mentioned previously, a block device can also be accessed like a character device. In that case, the I/O subsystem treats it as a character device, and bypasses the block cache. If a block device is accessed via both block- and character-device interfaces, applications may get inconsistent data from the device.

Data Cache

Data from block devices are accessed one block at a time. The I/O subsystem implements one or more data caches for all block devices. Once a block of

Scanned by CamScanner

data is read, it is kept in the cache on the assumption that the data will be requested again shortly. When the subsystem receives a block access request, it searches the cache to see if the request can be satisfied from the cache. In case of a cache hit, the request is executed immediately. Otherwise, it reads the block from the appropriate device. We do not discuss block cache here. We study cache management in detail in Chapter 14.

10.5 Disk Storage

In modern computers, disks are widely used as online secondary storage devices and swap devices. The most commonly used type is the magnetic disk. The other major types are magneto-optical disks, floppies, and CDs. Magnetic disks (disks, for short) are used to store persistent operational- and frequently accessed data. Disks can transfer several megabytes of data per second. The I/O subsystem, discussed in the previous section, is responsible for scheduling read-, write-, and other operation requests on disks. Some computer systems, to enhance performance and fault-tolerance, use logical disks that are an array of independent disks (RAID).

≫ A disk can be used as a raw device or a file system. For the latter option, data is stored in disks in the abstraction of files. The users of these data need have no worry about how to access them. It is for the file management system to make the data available in the main memory for their use. We discuss the file system in Chapter 11. Until then we assume the disk is used as a raw device.

10.5.1 Disk Geometry

To understand how disks work, we need to understand their physical structure. Figure 10.7 presents a model of a typical disk. A disk consists of many flat circular metallic plates called *platters*. Typically disks can have 1–12 of them. Platters are made of highly polished glass or ceramic materials coated with a fine layer of iron oxide. Platter diameters normally vary between 1.0

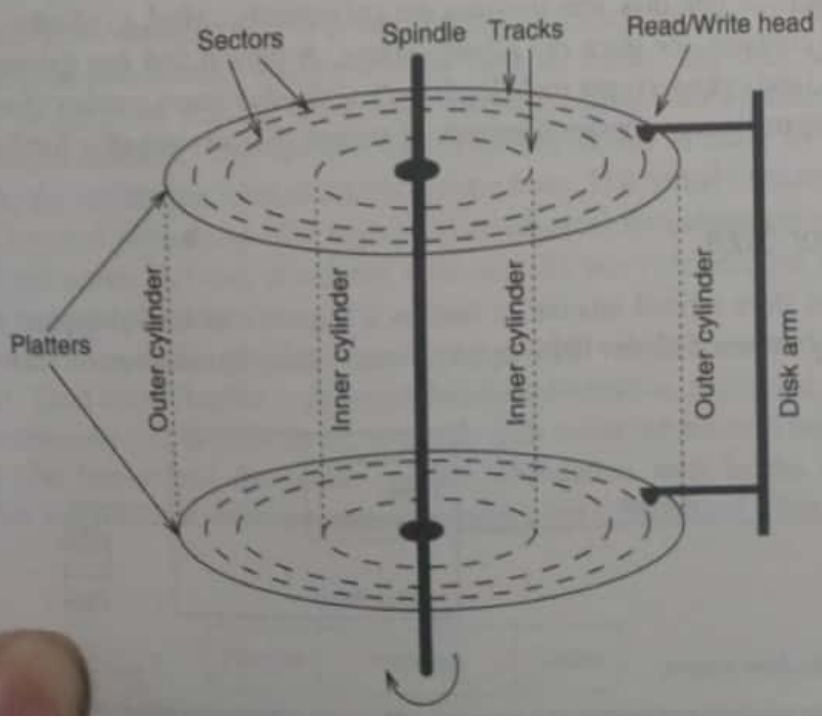


Figure 10.7: *Model of a disk.*

Scanned by CamScanner

and 5.25 in. Both surfaces of each platter are coated with magnetic materials that can store- and retain data in magnetically coded form. The platters are stacked on a single spindle. When a disk is in operation, the platters rotate at a high, but constant speed. This speed varies between 3600 and 7200 rotations per min depending on the model of the disk. For each platter surface, there is a read-write head that is responsible for reading- and writing data. The read-write heads do not physically touch the surface of platters; instead they float on a very thin cushion of air, the separation being about 10 millionths of an inch. All these heads are attached to a single arm, and they are all aligned in a line. The arm moves horizontally back and forth to position the heads on platter surfaces, and thereby, all read-write heads move together in unison. Note that disk heads do not read or write in parallel.

10.5.2 Disk Drive

The electronics that controls disk activities (such as moving the arm, and reading and writing blocks) directly is called *disk drive*, see Fig. 10.8. A disk controller is built into the disk drive. The disk controller is connected to an I/O controller through an I/O bus. The disk controller has to operate the disk drive to carry out operations on the disk. The controller has a built-in cache. At the lowest level, data transfer happens between the cache and the disk's recording surface.

≫ The number of sectors per track is normally not a constant number. Outer tracks hold more sectors than inner tracks. Modern disks have different zones containing a number of cylinders. Tracks in a zone have the same number of sectors.

10.5.3 Disk-space Organization

Before we can use a disk to store data, we have to organize the space available on the platters. Each platter surface is partitioned into a number of concentric circular tracks. A *track* is a thin circular stripe of area on a platter surface, (see Fig. 10.7). Tracks are concentric rings centred on the spindle. All the tracks at one disk arm position are collectively called a *cylinder*. A cylinder contains one track per platter surface. A typical disk has thousands of tracks (and cylinders) per inch. Each track is divided into a number of sectors, and the track may contain hundreds of sectors. Sectors are of a fixed size.

Sector Size

Sectors store a fixed amount of data as a sequence of bits. A typical sector usually stores 512 or 1024 bytes of user data. In addition, it can have

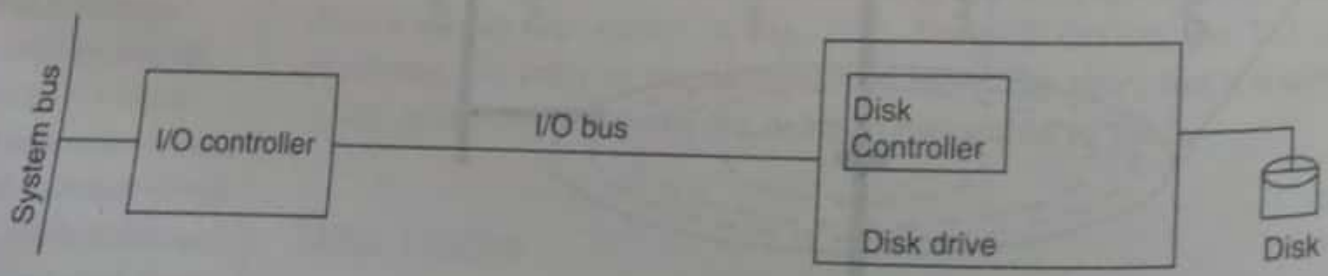


Figure 10.8: *Connecting a disk to the host system.*

Scanned by CamScanner

sector management information such as error detection and correction code. Sector size is a characteristic of the disk. The size is set when the disk is manufactured, and cannot be changed later. The sector is the smallest unit of data that can be written to or read from the disk surface. It is also the smallest unit of data transfer between the disk controller's built-in memory and the disk's recording surface.

Sector Address

A sector is identified by its cylinder number, the track number within the cylinder, and the ordinal position number within the track. Cylinders are normally numbered starting with 0 consecutively from the outermost cylinder to the innermost cylinder, and tracks are numbered starting with 0 consecutively from the top track to the bottom track, and sectors are numbered starting with 0 consecutively counter-clockwise starting from a reference position. That is, each sector is uniquely addressed by a tuple ⟨cylinder number, track number, sector number⟩.

10.5.4 Disk Formatting

A manufactured disk is in a random state. Before a disk can be used it is initialized into a usable state. The initialization is done in three steps: (1) physical formatting or low-level formatting, (2) partitioning, and (3) logical formatting or high-level formatting. These steps are discussed in the following subsections.

Physical Formatting

The disk when manufactured is a single uniform storage surface. It must be first divided into smaller basic elements such as tracks, and then sectors within the tracks so that the basic units (sectors) can be easily located and individually accessed. The division of tracks and sectors is made by magnetizing the surface particles along the chosen lines. This initial formatting is called *physical formatting*. The objectives of physical formatting are to: (a) locate and access the disk in smaller units and (b) serve consecutive sector access requests efficiently.

At the time of this physical formatting, special values are stored in all sectors. Each sector begins with sector header information, a payload space that is normally 512 or 1024 bytes, and ends with trailer information (see Fig. 10.9). The header and the trailer contain information such as the sector number and the error detection and correction code (checksum). Disk users

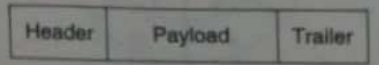


Figure 10.9: *Disk sector structure.*

Scanned by CamScanner

only access data in the payload area. The header- and the trailer contents are prepared by the disk controller, and are not visible to users. When a sector is written by a user, the disk controller prepares and writes the appropriate control information in the header and the trailer. The checksum is calculated using the payload content. When the sector is read, the checksum information is used to determine whether the sector content has become stale.

After dividing the disks into tracks and sectors, the sectors must be addressed suitably to serve the access requests for consecutive sectors efficiently. In order to access two consecutive sectors continuously in the same rotation (this is frequently the case), enough gap must be left between two adjacent sectors. Because the disk rotation is continuous, accessing adjacent sectors one after another would be impossible if there were no gap between the sectors to accommodate the computation delay. The gap left between sectors is called the *inter-sector* gap. There are many strategies used to address the disk sectors and we look at some basic strategies.

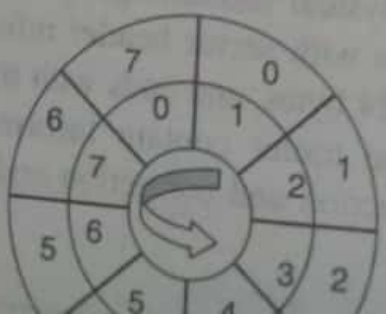
First, consider the case for accessing adjacent sectors. Assume that the sectors are addressed consecutively starting from a random location on the innermost track. After addressing all sectors in the current track, the same addressing strategy is applied to the next track, as shown in Fig. 10.10. The arrow shown indicates the direction of rotation of the disk, that is, anticlockwise. In this scheme, the inter-sector gap may not be good enough to assure the access of adjacent sectors (on the same track) in the same rotation. Assume that the disk controller has a limited size buffer, typically the case with most controllers. It is quite possible that continuous access to adjacent sectors in the same rotation might result in loss of data if the buffer is full. Data loss can be avoided by an extra rotation to empty the buffer, which is often too much of delay. Sufficient delay can be achieved by avoiding the addressing of sectors consecutively. For example, one full sector may be skipped between consecutive accesses. The addressing scheme of allowing one sector skip is shown in Fig. 10.11.

In general, to allow sufficient delay between consecutive disk accesses, there must be a fixed number of sectors between consecutive addresses.

0, 1, 2, 3, 4, 5, 6, 7 – consecutive addressing (shown in Fig. 10.10)

0, 4, 1, 5, 2, 6, 3, 7 – one sector distance (shown in Fig. 10.11)

This addressing scheme that leaves a number of sectors in-between is called *interleaving* and the number of sectors between two consecutive



Scanned by CamScanner

Figure 10.11: *Assigning addresses to sectors by one sector interleaving.*

addresses is referred to as the *level of interleaving*. The level of interleaving could be single (one sector distance), double (two sectors distance), etc.

Disk rotation and arm movement are the two independent mechanical actions that occur during disk access. The inter-sector gap and interleaving are aimed only at alleviating the delay in access due to disk rotation. Skewing is a strategy used to alleviate the access delay due to the arm movement between tracks. Assume that the disk rotates 2 sectors while the arm moves to the next track. Here, to allow continuous disk accesses in the same rotation, the address in the next track should start after 2 sectors in the next track as shown in Fig. 10.12. In the figure, after reading sector 7 from the inner track when the read-write head moves to the outer track, it is precisely at the beginning of the sector 0, and it can read/write the sector in the same rotation.

≫ Inter-sector gap, interleaving, and skewing are the addressing techniques used to enhance the efficiency of disk access.

This way of addressing the disk sectors in the adjacent tracks is called *skewing* and the *skewing distance* is defined as the number of sectors between the same address in adjacent tracks. For example, the skewing distance for the addressing scheme of Fig. 10.12 is 2.

In a normal disk, the arm movement is slower than disk rotation and hence the skewing distance is greater than the level of interleaving.

≫ As far as disk block/sector access is concerned, the I/O subsystem considers the entire disk a *single* device, and not a collection of partitions. The disk driver does not see partitions either. Partitions are seen at higher levels and these manage the partition space.

Partitioning

After the low-level physical formatting of a disk (done at the manufacturing site), we need to do another round of formatting of the disk at a higher level to make it usable by the operating system. This is called logical formatting, and it is discussed in the next section. Before logical formatting, one may create partitions on the disk space, where each partition contains a group of

Figure 10.12: *Linear sector addressing with skewing distance 2.*

Scanned by CamScanner

consecutive cylinders. Each disk has at least one partition. As far as the device space management is concerned, each partition is treated as a single hardware device (minidisk or logical unit). For example, a partition can hold a file system or a swap system. (The 0^{th} block in each partition is generally used to store primary bootstrapping information, and it is called a boot block.) The partition information is kept in a partition table that resides in the partition block of the first partition. The partition block follows the boot block, as shown in Fig. 10.13. Each entry in the partition table stores the starting cylinder number and information on the partition size. The rest of the blocks in the partition are used by a file system or used as swap space.

Logical Formatting

» Dividing a physical device into many non-overlapping partitions has several benefits. Different partitions can support different file systems (or swap space in virtual memory systems). If data in a partition is corrupted, the whole device does not become useless; the other partitions can be safely used.

After partitioning a disk, the operating system initializes its own data structures in each partition so that the disk becomes usable by the operating system. This is called logical formatting. At this juncture the operating system initializes disk sectors, at their payload spaces. The purpose of logical formatting is to create an empty file system in each partition which then allows the operating system to store and access files. We will see what constitutes an empty file system in Chapter 11. Since different operating systems use different file systems, the logical formatting is operating system-dependent. If the disk is formatted with a single file system, then it automatically restricts the number and types of operating systems that can be installed in the system. If one wants to use different operating systems with different file systems, then those file systems need to be created. This is usually done by first creating partitions in the disk (which is essentially creating logical boundaries within the disk) and then creating a suitable file system in each partition.

» Sector size determination, addressing, and partitioning are the key activities of low-level disk formatting. Recording boot block, file management structures (e.g. FAT, inode, etc.), directory, etc., are high-level disk formatting.

10.5.5 Disk Access

Disk space is logically viewed as a finite sequence of fixed size *blocks*. (The operating system stores data in the disk in units of blocks.) Each block is a contiguous sequence of finite data items, usually 1024 or 4096 bytes in length, that reside on the same track. When a disk is formatted, the disk block size may be set to a multiple (a power of 2) of the sector size. A block is also the unit of space allocation and deallocation by the operating system. The blocks are mapped onto physical disk sectors, and the disk geometry and the sector

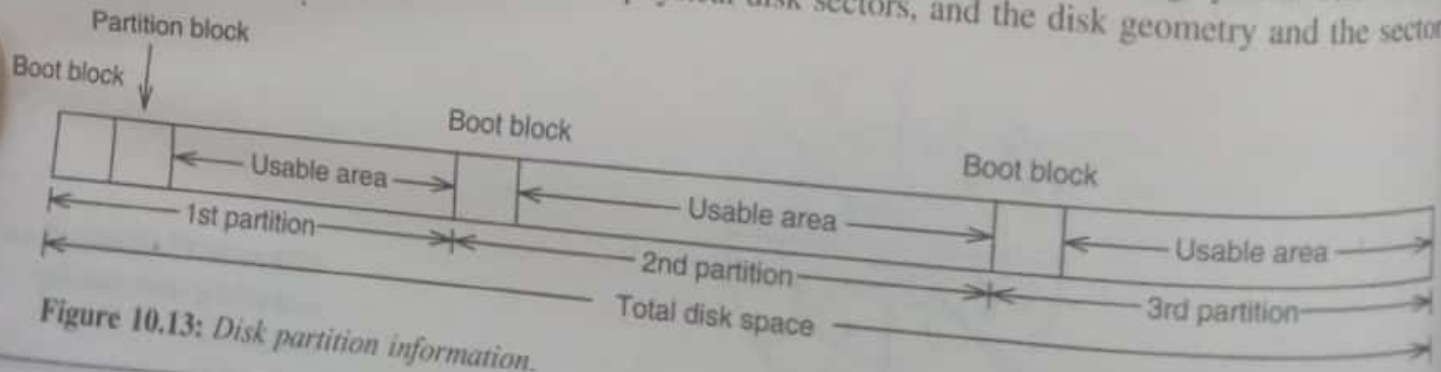


Figure 10.13: *Disk partition information.*

Scanned by CamScanner

addressing scheme help in the mapping. Block 0 is mapped starting with 0th sector on 0th track on 0th cylinder. The mapping continues in this order through 0th track, then through other tracks on the 0th cylinder, and then through other cylinders.

The CPU cannot directly access individual bytes in a disk block unless the block is copied into the main memory. The disk controller transfers data to the host system in units of blocks. Even if the CPU needs a few bytes from a disk block, the whole block is transferred. The CPU provides the block's tuple address to the disk's I/O controller and that, in turn, makes the whole block (all its sectors) available in the main memory. Alternatively, with smart disk controllers, the CPU may inform the controller the ordinal number of the logical block, and the controller translates the ordinal number to the tuple address(es) of the sectors. In either case, three time components are involved in a disk access. First, time is required to position the disk arm to the correct cylinder, and it is called *seek time*. Seek time is typically 5 to 25 ms on an average in modern disks. Second, time is required for the read-write head to be positioned at the beginning of the first sector of the block, and is called the *rotational latency time*. (The time is 8 to 16 ms per complete rotation.) Third, time required to transfer the data between the disk and the main memory (via the disk controller internal memory), and is called *transfer time*.

A disk drive directly controls the disk hardware circuit. The disk drive connects to a computer through a host I/O controller. For example, in the SCSI domain, a HBA can have many disk drives connected to it through an SCSI bus (see Fig. 10.2 on page 278). To perform a disk I/O operation, the CPU instructs the HBA what to do. The HBA, in turn, communicates with the corresponding disk drive adhering to SCSI protocols. They exchange SCSI commands, messages, data, and status information to execute the disk I/O request. For example, consider a typical disk read request. The CPU provides the HBA the target disk number, and the sector tuple address. The HBA instructs the corresponding disk drive to read the sector. The disk drive re-positions the read-write head and moves the required sector data from the disk surface into the local buffer first, and next transfers this local copy to the host HBA's internal memory buffer over the SCSI bus. The DMA device on the HBA board transfers this data to the main memory. The CPU does not have to know the SCSI bus architecture and bus protocol to be able to access the disk.

≫ A disk can transfer 2 to 50 MB data per second. Thus, the transfer time is negligible compared to the seek- and latency times.

Most modern disk controllers include on-board memory to buffer data from disks. The buffer usually holds a complete track at a time. The controller reads disk content track-by-track. Upon reading the track, the disk controller transfers data from/to host system one block at a time. As stated in Section "Data Cache" on page 284, some systems also cache these blocks at a higher level of software hierarchy.

≫ Unless there are requests in the disk queue, scheduling algorithms are useless. Thereby, they are more useful in high-end systems such as servers than in personal computers.

10.5.6 Disk Scheduling

As mentioned previously, there are three time factors involved in a disk access: the seek time, the rotational latency time, and the data transfer time. A disk scheduler's objective is to optimize average access time. When an access request

Scanned by CamScanner

is made on disk, and if the disk drive and disk controller are available, the request can be carried out immediately. Otherwise, the request needs to wait in a device queue, (see Fig. 10.6 on page 284). When the controller and the device become free, one of the waiting requests is served. A disk scheduler determines the request to be served next. Many disk scheduling algorithms have been investigated in the past, most of them are oriented towards optimizing the seek time. A few of them are discussed in the following subsections.

We take this disk access request string as our reference—25, 125, 75, 175, 100, 50, and 150—to describe examples for the scheduling schemes, where the numbers are cylinder numbers. The disk has 200 cylinders, numbered 0…199, from the outermost cylinder to the innermost cylinder. We assume that at time 0 the read-write head is at the middle cylinder (100) and is moving toward the innermost cylinder (199).

FCFS Scheduling

As the name suggests, access requests are served by the scheduler on their arrival order. Figure 10.14 depicts the read-write head movements for the FCFS scheduling of our reference request string. This is a very simple scheduling discipline, and is easy to implement. This discipline, however, does not perform well as head movements are too many, and sometimes there are wild swings between the inner- and outer cylinders skipping requests that fall onto the middle cylinders. As shown in the figure, the read-write head moves from cylinder 25 to 125 without servicing the requests on the middle cylinders 50, 75, and 100, leading to a quite unnecessary delay in their service. For this example request string, the total number of head movements is 550 cylinders.

Shortest-seek-time-first (SSTF) Scheduling

Disk head movement is a time consuming operation. We need to reduce the number of head movements as far as possible. A reasonable scheduling

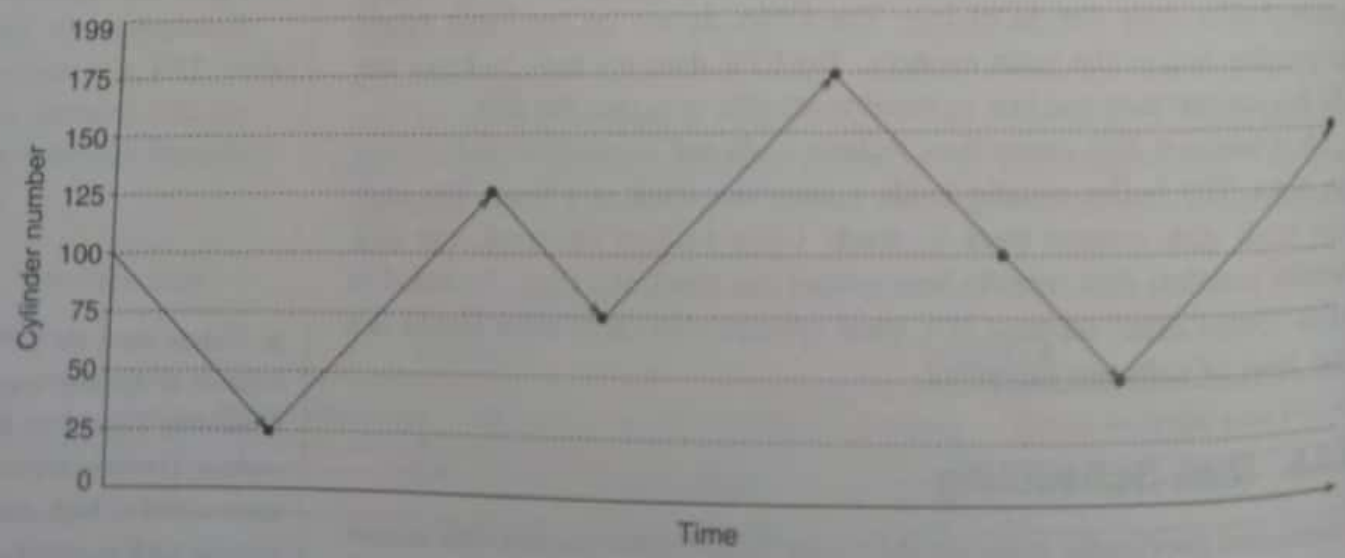


Figure 10.14: *FCFS scheduling of requests on cylinders 25, 125, 75, 175, 100, 50, and 150.*

Scanned by CamScanner

discipline is to serve requests nearest to the current head position. Figure 10.15 depicts the read-write head movements for SSTF scheduling of our reference request string. This discipline displays very good average performance, but is not optimal. Moreover, it suffers from the starvation problem. Starvation occurs when requests continuously pour into the neighbourhood of the current head position while some requests are waiting at far off cylinders. For the example reference string, the total number of head movements is 225 cylinders.

Scan, C-scan, and Elevator Scheduling

In scan scheduling, the read-write head moves from one end to the other end. While moving from one cylinder to the next, it serves pending requests on these cylinders. When the farthest end on the direction is reached, the direction of the head movement is reversed and it starts servicing new requests. Thus, the disk arm (acts as a shuttle service in transportation and) shuttles back and forth across all cylinders serving requests. In scan scheduling, middle cylinders get preferential treatment compared to outer ones. A variation of scan scheduling is the circular scan called *c-scan* where the head returns to the first cylinder immediately after it reaches the last cylinder, without serving requests in the reverse direction. In c-scan, the service is more uniform.

≫ A variant of c-scan is typewriter, in which the arm returns to cylinder 0 after servicing requests on the farthest cylinder.

In practical implementation, the read-write head does not always move from the outermost cylinder to the innermost cylinder, and vice versa. Like elevators in buildings, the head oscillates between final requests in each direction of head movement. Such scheduling is called *elevator scheduling* or *look scheduling*.

Figure 10.16 depicts the read-write head movements for scan-, c-scan-, and elevator scheduling algorithms of our reference request string. For the example, the total numbers of head movements are 275, 375, and 225 cylinders, respectively for the scan-, c-scan-, and elevator scheduling algorithms. These scheduling schemes do not suffer from starvations.

Scheduling in Hardware

In the disk scheduling schemes described above, we have tried to optimize the seek time. There is another factor involved in servicing a request, namely

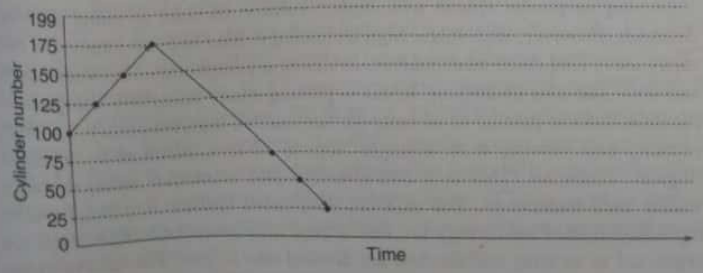


Figure 10.15: *SSTF scheduling of requests on cylinders 25, 125, 75, 175, 100, 50, and 150.*

Scanned by CamScanner

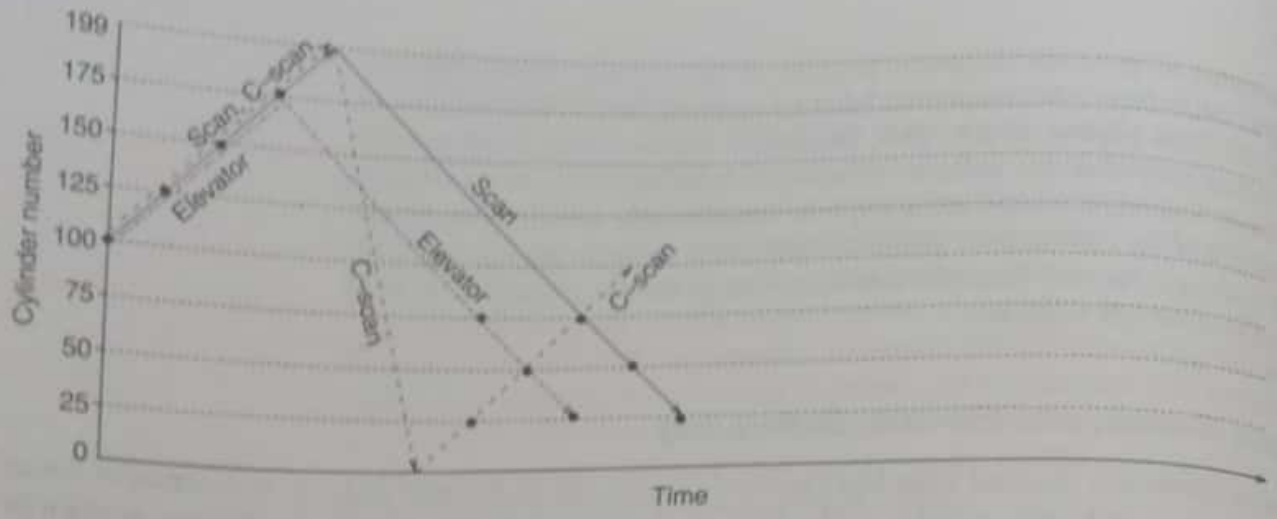


Figure 10.16: *Scan, c-scan, and elevator schedulings of requests on cylinders 25, 125, 75, 175, 100, 50, and 150.*

the *rotational latency*. In general, it is difficult for the operating system to schedule requests ordered by block positions to reduce rotational latency delays. The disk continuously spins around its spindle. So, it is difficult to pinpoint the current head position, and the disk drive does not disclose the positional information to the operating system.

Many modern disk controllers have an embedded disk-scheduling algorithm. The operating system blindly dumps all requests on the controller, and the controller schedules them the way it wants. The controller can exploit rotational latency, and produce a better schedule than that of the software disk scheduler. Sometimes, however, the operating system may not prefer the order of service chosen by the disk controller. For example, when a file operation modifies many blocks, we may need to write the blocks in a particular order. As another example, kernel-initiated I/O operations such as page swap in demand paging systems should get the highest priority. In such situations, the controller may need to blindly obey the kernel order without doing any local optimizations.

10.5.7 Redundant Array of Inexpensive/Independent Disks (RAID)

≫ RAID originally stood for redundant array of inexpensive disks. It was later renamed, in the computing industry, as redundant array of independent disks. Its objective was a higher level of performance, reliability, and data volume using an array of inexpensive disks.

As for any other computing components, the technology of disks has been improving to greater speeds with higher volume of data to meet growing demands. However, such single high technology disks are too expensive and many users find them unaffordable. RAID is a compromise between speed and cost. The idea underlying RAID is placing many relatively inexpensive identical disks in parallel as an array for fast access, high reliability, and high capacity and viewing it as a single 'logical' disk controlled by a single 'logical' controller. This is a cost-effective and attractive solution and a widely used technology. The data are distributed (called *stripping*) and/or replicated (called *mirroring*) for fast access and reliability. The disks are organized as an array and the data are divided into stripes. The unit of stripe

Scanned by CamScanner

could be a bit, byte, word, sector, etc. There were five combinations initially suggested to store and access these stripes in the array of disks and they are called RAID levels.

RAID Level 0: Store the stripes across the array of disks in round-robin fashion and access the disk units in parallel. Here the stripe size is 1 bit. This scheme uses stripping without parity.

RAID Level 1: Level 0 is mirrored to achieve fault tolerance. That is, disk array is duplicated and the stripes are written on both arrays and read from one array. This scheme uses mirroring without parity.

RAID Level 2: Stripe the data into 4 bits and make a 3-bit Hamming code to create a 7-bit word. This can correct one bit-errors and detect two bit-errors. Store each 7-bit word in seven disks (one bit per disk) and access them parallelly. This scheme increases reliability with the use of the Hamming code.

RAID Level 3: Use a one parity bit instead of 3-bit Hamming codes, as in level 2, and use five disks instead of seven to store these five bits. It is a simplified version of level 2. This scheme increases reliability through the use of parity.

RAID Level 4: With level 3 organization, increase the stripe size. This scheme increases the size of parity to the size of a block.

RAID Level 5: With increased stripe size, distribute the parity stripes in round-robin fashion. That is, in a disk array of size k, store the first parity stripe in disk k, second parity stripe in disk $k-1$, third parity stripe in disk $k-2$, etc. This scheme distributes the block level parity across the disk to increase reliability.

It is easy to see that each organization has its advantages and limitations and, therefore, suitable for various applications.

10.6 The Network Interface Card

Modern computers generally form part of computer networks. The computers in a network communicate with each other. Hardware devices facilitate inter-computer communications. Each computer system has some network interface cards (NICs) connected to it, and the NICs help in the actual data transmission. NICs include Ethernet, token ring, bluetooth, and many others. A NIC is a special I/O device that puts outgoing data on a network line, and retrieves incoming data from the line. In this section, we briefly study DP83820, an Ethernet device from National Semiconductors. The DP83820 is a single-chip 10/100/1000 megabits per second Ethernet controller for PCI bus architecture. Figure 10.17 presents a block diagram of the DP83820. It consists of PCI bus interface, BIOS ROM, EEPROM interfaces, receive and transmit data buffer managers, an 802.3 media access controller (MAC), SRAM, DMA device, management information base (MIB), and receive filter support logic. It implements a 66 megahertz, 64-bit PCI bus interface for host communications. It can support full duplex transmission and reception on the physical interface.

≫ Full duplex operatio
the simultaneous
transmission and recep
of packets.

Scanned by CamScanner

Two independent MAC units perform the control function for the media access of transmitting and receiving packets to and from an Ethernet. The DP83820 has two independent FIFOs; one is used by the receive buffer manager and the other by the transmit buffer manager. The FIFOs act as temporary storage of Ethernet packets so that the host system is free from real-time demands from the network. The transmit FIFO is 8 Kbits, and the receive FIFO is 32 Kbits. Each FIFO is associated with a threshold value. When the transmit FIFO falls below or the receive FIFO crosses its threshold value, the DP83820 transfers data from or to the main memory. The transmit buffer manager reads packet data from the main memory, places the data in the transmit FIFO, and sends the data to the transmit MAC. The receive buffer manager retrieves packet data from the receive MAC, places the data in the receive FIFO, and writes the data in the main memory. The receive filter logic allows software control for accepting packets based on destination addresses and packet types. The address recognition logic includes supports for broadcast-, multicast-, and unicast addresses.

The transmit MAC implements the transmit portion of the 802.3 media access control. The MAC retrieves packet data from the transmit buffer manager and puts the packet on the physical layer device interface. The MAC also provides MIB control information for transmit packets. The receive

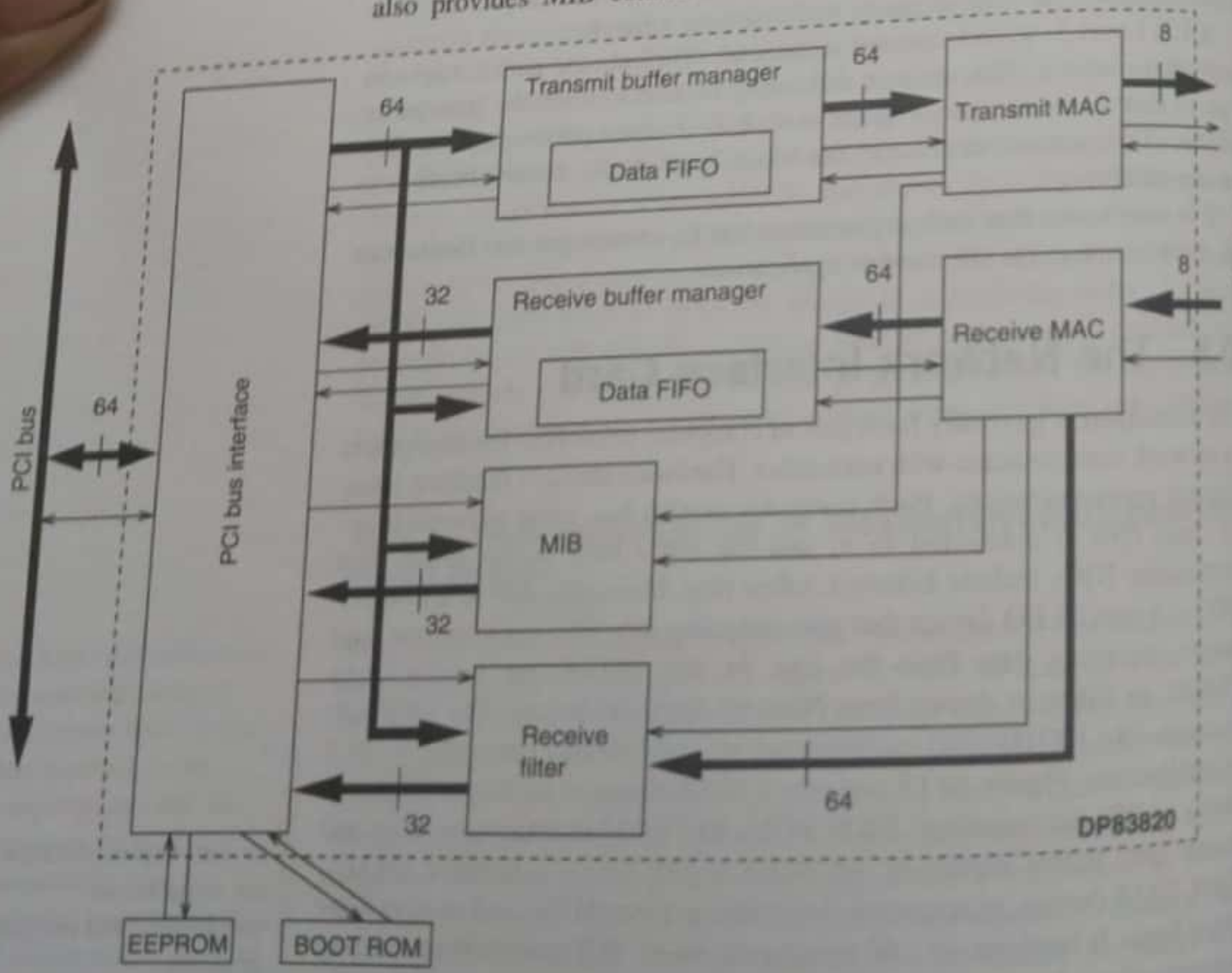


Figure 10.17: *A model of the DP83820 gigabit Ethernet NIC.*

Scanned by CamScanner

MAC implements the receive portion of the 802.3 media access control. The MAC retrieves packet data from the receive portion of the physical layer and sends the packet to the receive buffer manager. The MAC also provides MIB control information for the receive packets.

10.6.1 Ethernet Packet Format

Figure 10.18 presents the standard format of Ethernet packets. Each Ethernet device has a unique number or address assigned to it for the data link layer addressing. Ethernet addresses are 6-byte long. Two devices on an Ethernet use their Ethernet addresses for communications between them. Each computer in a network also has a unique host identifier. Periodically, the operating system generates a packet containing the host identifier and the Ethernet number, and broadcasts the packet over the Ethernet. Broadcasts use the special Ethernet address (0xFFFFFFFFFFFF). When another host receives the packet, it recognizes that the sending host is on the same Ethernet with a specific NIC card having the said Ethernet number. These broadcast packets help the computers on the same Ethernet to know one another (their host identifiers and NIC addresses).

≫ The unique number is built into each Ethernet device at the time of manufacture. It is guaranteed to be unique across all Ethernet NICs developed by all its vendors.

10.6.2 Packet Descriptor

A packet of data is represented by one or more packet descriptors. A descriptor comprises the following components: (1) a link to next descriptor, (2) a pointer to a buffer that contains the data, (3) buffer size, and (4) status. Figure 10.19 shows a typical DP83820 descriptor. There is a bit in the status (called MORE) that indicates whether the packet is single descriptor or is split into multiple descriptors. If the MORE bit is 0, it is a single descriptor packet. If the bit is 1, the packet has more descriptors that can be obtained using the link component. The terminating descriptor has 0 as the MORE bit value, as shown in Fig. 10.19(b). The software (device driver)

≫ Ethernet is a typical physical communication medium that physically connects two or more computers. It physically exchanges messages among computers. The well-known TCP and IP are communications protocols for the exchange of messages at a higher layer of abstraction. Higher-level protocols help processes meaningfully exchange data among themselves independent of the physical characteristics of transmission mediums.

Preamble	10101010
Start of frame	10101011
Destination address	Destination Ethernet number or broadcast address
Source address	Local Ethernet number
Data length	Protocol identifier/Data length
Data	Data + pad >= 64 bytes
Pad	
Checksum	

Figure 10.18: *Standard Ethernet packet format.*

Scanned by CamScanner

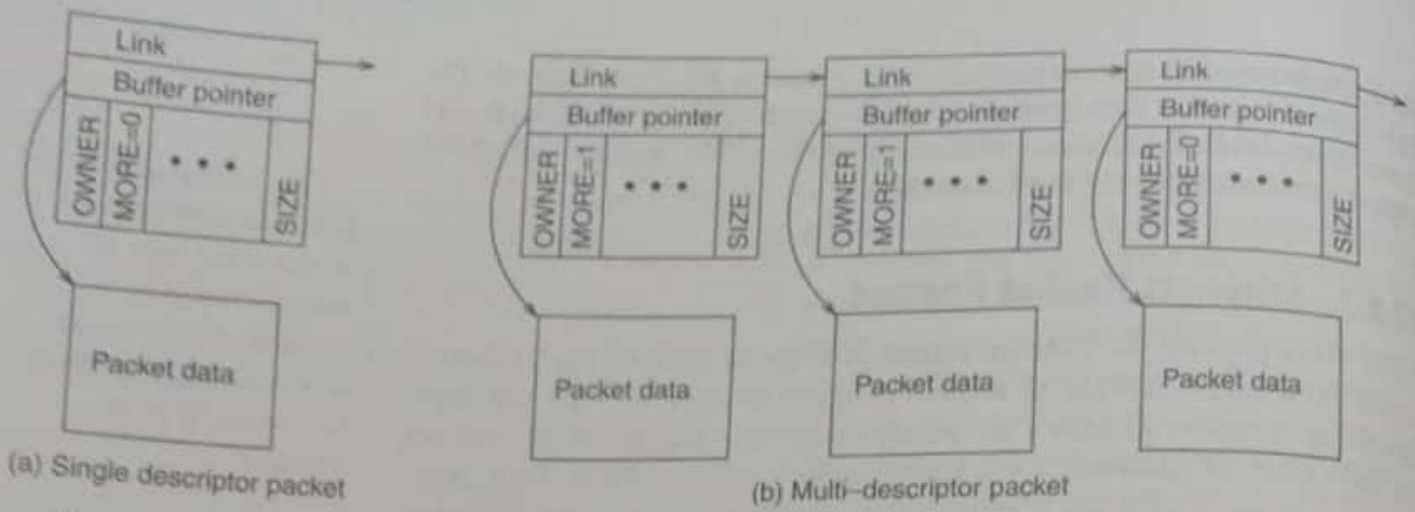


Figure 10.19: *The DP83820 packet descriptor.*

registers descriptor addresses with the NIC via its I/O ports. The transmit manager scans transmit descriptor(s) and forms an Ethernet packet in the transmit FIFO using their buffer contents. The receive manager splits an incoming packet into buffers identified by receive descriptors. Data transfer between the main memory and the NIC is done using the on-board DMA device, thereby reducing the burden on the main CPU. There is another bit in the status (called OWNER) that identifies the current owner of the descriptor. This bit helps in recycling descriptors. Descriptors are initially owned by the producer of the data (the device driver for transmit descriptors, and the NIC for receive descriptors).

10.6.3 Software–Hardware Interaction

For a thorough understanding of software-hardware interactions, let us consider a typical message sending operation. When a process wants to send a message to a particular host on the network, the process provides the destination host identifier to the local operating system. The message is passed on to the NIC driver. The driver forms Ethernet packets out of the message and passes them on to NIC hardware for transmission. The driver allocates packet descriptors to the packets, and initializes the descriptor status bits. It then makes the NIC the owner of the descriptors, and registers with the NIC the descriptor address(es). After transmission of the packets, the NIC makes the driver the owner of the descriptors, and interrupts the CPU to convey that the packet transmissions are over.

During an Ethernet packet transmission, the preamble, the start of frame, and the checksum contents are added by the NIC. See Fig. 10.18 on page 297 for the Ethernet packet format. (During a reception, the preamble- and start of frame components are stripped by the NIC.) Each packet has a component for a destination Ethernet number. The driver or a higher level software component has to locate the destination Ethernet number. If the destination host is on the same Ethernet, the sending system finds the remote Ethernet card

Scanned by CamScanner