9

Input/Output Management

Learning Objectives

After reading this chapter, you will be able to:

- Discuss principles of I/O hardware including I/O devices, device controllers, and DMA.
- List different goals of the I/O software.
- Explore different ways to perform I/O operations.
- Describe the functions of each layer in the I/O software.

9.1 INTRODUCTION

A computer system contains many I/O devices and usually, most of the computer time is spent in performing I/O. Thus, controlling and managing the I/O devices and the I/O operations is one of the main responsibilities of an operating system. The operating system must issue commands to the devices to work, provide a device-independent interface between devices and the rest of the system, handle errors or exceptions, catch interrupts, etc.

Since a variety of I/O devices are attached to the system, providing a device-independent interface is a major challenge for operating system designers. To meet this challenge, designers use a combination of hardware and software techniques. Most I/O devices are connected to ports, buses, or device controllers. To hide the details of different devices, operating system designers let the kernel use device-driver modules. Note that device drivers present a uniform device-access interface.

9.2 PRINCIPLES OF I/O HARDWARE

A computer communicates with a variety of I/O devices ranging from mouse, keyboard, and disk to highly specialized devices like a fighter plane's flying controls. However, one needs to digest only a few concepts to understand how operating systems facilitate device-independent communication with I/O devices.

9.2.1 I/O Devices

As stated earlier, a wide variety of I/O devices are attached with a computer system. All these I/O devices can be roughly divided into two classes: *block* and *character* devices. A **block device** stores data in fixed-size blocks with each block having a specific address. The data transfer to/from a block device is performed in units of blocks. An important characteristic of block devices is that they allow each block to be accessed independently, regardless of other blocks. Some commonly used block devices include hard disks, USB memory sticks, and CD-ROMs. Applications can interact with the block devices through the **block-device interface**, which supports the following basic system call:

- **read()**: To read from the device.
- **write()**: To write to the device.
- **seek()**: To specify the next block to be accessed.

On the other hand, a **character device** is the one that accepts and produces a stream of characters. Unlike block devices, character devices are not addressable. The data transfer to/from them is performed in units of bytes. Some commonly used character devices include keyboards, mice, and printers. Applications can interact with a character device through the **character-stream interface**, which supports the following basic system calls.

- **get()**: To read a character from the device.
- **put()**: To write a character to the device.

In addition to block and character devices, there are certain devices that do not fit under any of these categories. For example, clocks and timers are such devices. They are neither block addressable nor they accept or produce character streams; rather they are only used to generate interrupts after some specified interval.

clock & timer

> *Note: Though block devices allow random access, some applications (for example, DBMS) may access the block device as a sequential array of blocks. This type of access is referred to as* ***raw I/O.***

An I/O device is attached with the computer system via a connection point known as a **port** (like serial or parallel port). After being attached, the device communicates with the computer by sending signals over a bus. A **bus** is a group of wires that specifies a set of messages that can be sent over it. Recall that Figure 1.2 of Chapter 01 shows the interconnection of various components to the computer system via a bus. Note that the connection of some devices to the common bus is via device controllers.

9.2.2 Device Controllers

I/O units consists of a mechanical & electronic compo-t
(device itself) (device driven)

Imp

A device controller (or **adapter**) is an electronic component that can control one or more identical devices, depending on the type of device controller. For example, a **serial-port controller** is simple and controls the signals sent to the serial port. On the other hand, an **SCSI controller** is a complex device that can control multiple devices.

Edc) chip on mother board or printed circuit card can be inserted on expansion slot.

Each device controller includes one or more registers that play an important role in communication with processor. The processor writes data in these registers to let the device take some action, and reads data from these registers to know the status of the device. There are two approaches to let the communication between processor and controller occur.

- In the first approach, special 1) I/O instructions are used, which specify the read or write signal, **I/O port address**, and CPU register. The I/O port address helps in the selection of correct device.
- The second approach is **memory-mapped I/O**. In this, registers of device are mapped into the address space of the processor and standard data-transfer instructions are used to perform read/write operation with the device. 2)

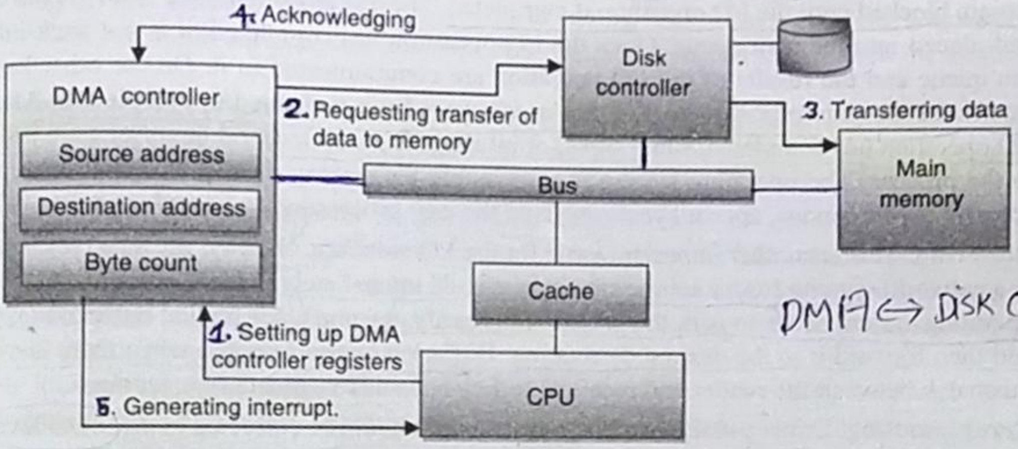
Imp

9.2.3 Direct Memory Access (DMA)

Devices like disk drives are frequently involved in transferring large amounts of data. To keep the CPU busy in transferring the data one byte at a time from such devices is clearly wastage of the CPU's precious time. To avoid this, a scheme called **Direct Memory Access (DMA)** is often used in systems. Note that for using DMA, the hardware must have a DMA controller, which most systems have. In DMA, the CPU assigns the task of transferring data to the DMA controller and continues with other tasks. The DMA controller can access the system bus independent of CPU, so it transfers the data on its own. After the data has been transferred, it interrupts the CPU to inform that the transfer is completed.

> ***Note:*** *Some systems have a separate DMA controller for each device whereas some systems have a single DMA controller for multiple devices.*

DMA works as follows: (1) The CPU tells the DMA controller the source of data, destination of data, and the **byte count** (the number of bytes to transfer) by setting up several registers of DMA controller. (2) The DMA controller then issues the command to the disk controller to read the data into its internal buffer and verifies that no read error has occurred. After this, (3) DMA controller starts transferring data by placing the address on the bus and issuing read request to disk controller. Since the destination memory address is on the address lines of bus, the disk controller reads data from its buffer and writes to the destination address. (4) After the data has been written, the disk controller acknowledges DMA controller by sending signal over the bus. The DMA controller then increments the memory address to use and decrements the byte count. If byte count is still greater than 0, the incremented memory address is placed over the bus' address lines and read request is issued to the disk controller. The process continues until byte count becomes 0. (5) Once the transfer has been completed, the DMA controller generates an interrupt to the CPU. The entire process is illustrated in Figure 9.1.



DMA ↔ Disk controller

Figure 9.1: Transferring Data using DMA

Note that when DMA controller acquires the bus for transferring data, CPU has to wait for accessing bus and main memory; though it can access cache. This mechanism is called **cycle stealing** and it can slow down the CPU slightly. However, it is important to note that large amounts of data get transferred, with negligible task, by the CPU. Hence, DMA seems to be a very good approach of utilizing CPU for multiple tasks.

9.3 PRINCIPLES OF I/O SOFTWARE

Hardware and software are like two sides of a coin: one is unusable without the other. Thus, in this section, we will first discuss the goals of I/O software and then different ways to perform I/O operations from the operating system's point of view.

There are basically three different ways of performing I/O operations: *programmed I/O*, *interrupt-driven I/O*, and *direct memory access (DMA)*. Since the DMA has already been discussed in the previous section, we will only discuss programmed I/O and interrupt-driven I/O in this section.

9.3.1 Goals of the I/O Software Imp.

The I/O software must be designed keeping the following goals in mind:

- **Device independence:** This is one of the major issues that should be considered while designing the I/O software. Device independence allows the programmers to write programs for performing I/O operations independent of the I/O devices. That is, they can write device-independent programs that can be used for any kind of I/O device without having to specify the device in advance. For example, if a programmer has written a program to read from a file, then it should be able to read from a file on hard disk, floppy disk, or CD-ROM. It is the responsibility of the operating system to handle the problems caused by the differences among I/O devices.

- **Uniform naming**: This is another important issue related to device independence. The names of devices must be symbolic, such as a collection of strings or integers, and independent of the device. The user should be able to access each device in a similar manner using the path name. For example, Linux integrates all I/O devices into a file system and each I/O device is assigned a path name, normally under the directory `/dev`. For example, a printer might be accessed as `/dev/lp`.
- **Synchronous vs. asynchronous I/O**: An operating system may support synchronous or asynchronous I/O. The synchronous (blocking) I/O causes the process (that requires I/O) to remain blocked until the I/O operation is completed. The process is removed from the run queue and placed into the wait queue. Once the I/O operation has completed, it is put back into the run queue and the results of the I/O operation are communicated to it. On the other hand, in asynchronous (interrupt-driven) I/O, the process need not wait for I/O completion. After the I/O operation has completed, some signal or interrupt is generated, and the results are provided to the process. The operating system is responsible for making those I/O operations that are actually asynchronous, appear synchronous to the user processes.
- **Buffering**: This is another important issue for the I/O software. Usually, the data (say, a packet in a network) coming from a source cannot be passed immediately to the destination. Hence, the operating system needs to park the packet temporarily at some place (called buffer), examine it and then forward it to the desired destination. Buffering is also required when there is a speed mismatch between the sender and receiver or they have different data transfer sizes.
- **Error handling**: Errors usually occur in a system and should be corrected in the lower levels (as close to the hardware as possible) without the upper levels knowing about them. For example, if device controller detects some error (say, a read error), it must try to correct it. If it cannot correct the error, then only should the error be propagated to upper level that is, to the device driver which should handle the error by repeating the read operation. However, only transient errors can be recovered by repeating the operation.
- **Dedicated vs. sharable devices**: Dedicated devices are those that can be used by only one user at a time and can be assigned to others only after the current user has finished with it. For example, a tape drive is a dedicated device. The sharable devices, on the other hand, can be used by multiple users simultaneously. For example, a number of users can have open files at the same time on a hard disk. The use of dedicated devices often leads to problems such as deadlock. It is the responsibility of the operating system to handle both types of devices in such a way that any kind of problem is avoided.

9.3.2 Programmed I/O

Programmed I/O is the simplest method to perform I/O, in which the CPU does all the work. A complete interaction between the host (CPU) and a controller may be complex, but the basic abstract model of interaction can be understood by a simple example. Suppose that a host wishes to interact with a controller and write some data through an I/O port. It is quite possible that the controller is busy performing some other task; hence, the host has to wait before starting an interaction with a controller. When a host is in this waiting state, we say the host is **busy-waiting** or **polling**.

Note: *Controllers are programmed to indicate something or understand some indications. For example, every controller sets a busy bit when busy and clears it when gets free.*

CPU⟷ I/O Device.

CPU

To start the interaction, the host continues to check the busy bit until the bit becomes clear. When a host finds that the busy bit has become clear, it writes a byte in the data-out register and sets the write bit to indicate the write operation. It also sets the command-ready bit to let the controller take action. When controller notices that the ready bit is set, it sets the busy bit and starts interpreting the command. As it identifies that write bit is set, it starts reading the data-out register to get the byte and writes it to the device. After this, the controller clears the ready bit and busy bit to indicate that it is ready to take the next instruction. In addition, the controller also clears an error bit (in the status register) to indicate the successful completion of I/O operation.

9.3.3 Interrupt-driven I/O

The above scheme for performing interaction between host and controller is not always feasible, since it requires busy-waiting for host. When either controller or device is slow, this waiting time may be long. In that scenario, the host must switch to another task. However, if the host switches to another task and stops checking the busy bit, how would it come to know when the controller becomes free?

(1) One solution to this problem is that the host must check the busy bit periodically and determine the status of the controller. This solution, however, is not feasible because in many cases, the host must service the device continuously; otherwise the data might be lost.

(2) Another solution is to arrange the hardware with which a controller can inform the CPU that it has finished the work given to it. This mechanism of informing the CPU about completion of a task (rather than the CPU having to inquire about completion of the task) is called **interrupt**. The interrupt mechanism eliminates the need of busy-waiting of processor and hence this solution is considered more efficient than the previous one.

Now let us understand, how the interrupt mechanism works. The CPU hardware has an interrupt-request line, which the controllers use to raise an interrupt. Controller asserts a signal on this line when I/O device becomes free after completing the assigned task. As CPU senses the interrupt-request line after executing every instruction, it frequently comes to know that an interrupt has occurred. To handle the interrupt, the CPU performs the following steps :

1. It saves the state of current task (at least the program counter) so that the task can be restarted (later) from where it has been stopped.
2. It switches to the interrupt-handling routine (at some fixed address in memory) for servicing the interrupt. The interrupt handler determines the cause of interrupt, does the necessary processing, and causes the CPU to return to the state prior to the interrupt.

The above discussed interrupt-handling mechanism is the ideal one. However, in modern operating systems, the interrupt-handling mechanism must accommodate the following features:

- High-priority interrupts must be identified and serviced before low-priority interrupts. If two interrupts occur at the same time, the interrupt with high-priority must be identified and serviced first. Also, if one interrupt is being serviced and another high-priority interrupt occurs, the high-priority interrupt must be serviced immediately by preempting the low-priority interrupt.
- CPU must be able to disable the occurrence of interrupts. This is useful when CPU is going to execute those instructions of a process that must not be interrupted (like instructions in the critical section of a process). However, disabling all the interrupts is not a right decision. This is because the interrupts not only indicate the completion of task by a device but many exceptions also such as an attempt to access non-existent memory address, divide by zero error, etc. To resolve this, most CPUs have two interrupt-request lines: *maskable* and *non-maskable* interrupts. Maskable interrupts are used by device controllers and can be disabled by CPU whenever required, but non-maskable interrupts handle exceptions and should not be disabled.

9.4 I/O SOFTWARE LAYERS

To achieve the goals of I/O software in an efficient manner, the I/O software is structured in four layers: *interrupt handlers*, *device drivers*, *device-independent I/O software*, and *user-level I/O software* (see Figure 9.2). Each of these layers performs a well-defined function and presents a well-defined interface to its neighbouring layers. The interfaces and functionality of layers may differ from system to system. In this section, we will discuss different layers of I/O software irrespective of any specific system.

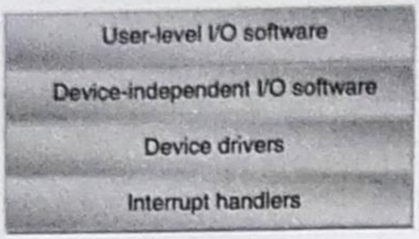


Figure 9.2: Layers in I/O Software

9.4.1 Interrupt Handlers

Though interrupt-driven I/O is more efficient than programmed I/O, for almost all I/Os, interrupts are displeasing yet cannot be avoided. However, they can be hidden away in such a manner that only a limited part of the operating system is aware of them. One way to hide interrupts is to let each process that starts an I/O operation block until the I/O operation has finished and the interrupt occurs. Note that a process can block by performing a `wait` operation on a semaphore or executing `wait()` on a condition variable in case of monitors or invoking a `receive()` call on a message in case of message passing. In each case, the net effect will be that the process will be blocked.

Whenever an interrupt is fired, the CPU stops executing the current task, and immediately transfers the control to the interrupt-handler routine at a predefined location in memory. The interrupt-handler performs everything required to unblock the process that initiated it. It can unblock a process by performing a `signal` operation on a semaphore or executing `signal()` on a condition variable in case of monitors or sending a message to the blocked process in case of message passing. Whatever operation the interrupt handler performs, the blocked process now becomes runnable.

9.4.2 Device Drivers

The operating system has to deal with a variety of I/O devices that can be attached with the computer system. However, it is almost impossible for an operating system developer to write separate code to handle every distinct I/O device. That is because then manufacturing of each new device would lead to changing or adding some code in the operating system. Clearly, this is not a feasible solution. Instead, I/O devices are grouped under a few general kinds. For each general kind, a standardized set of functions (called **interface**) is designed, through which the device can be accessed. The differences among the I/O devices are encapsulated into the kernel modules called **device drivers**. Note that a device driver is specific to each device and each device driver exports one of the several standard interfaces. Figure 9.3 shows the logical positioning of device drivers in the kernel space.

It is a program that controls particular type of device that is attached to your computer.

Ex- device driver for printers, displays, CD-ROM readers, diskdrives etc.

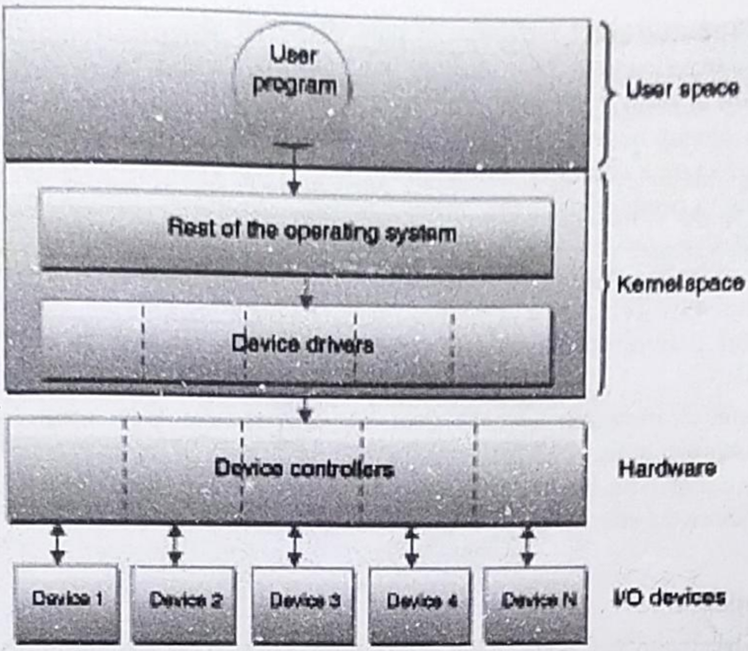


Figure 9.3: Logical Positioning of Device Drivers in Kernel Space

9.4.3 Device-Independent I/O Software

Though a part of I/O software (that is, device drivers) contains the device-specific code, other portions of I/O software are device independent. The device-independent I/O software layer is responsible for performing various functions such as uniform interfacing for device drivers, buffering, and error handling.

(a) Uniform Interfacing for Device Drivers

As discussed earlier, one of the important goals of I/O software is to provide a uniform interfacing to all the I/O devices and device drivers. This is required because if different devices are interfaced differently, the inclusion of every new device in the system would demand modification in the operating system. By providing a uniform interface, each device can be accessed through one of the standard interfaces in a manner independent of the device itself. Also, when hardware vendors manufacture new device, they can either make it compatible with one of the several available device drivers or write the new device driver exporting one of the several standard interfaces.

Another aspect related to uniform interfacing is how I/O devices are named. It is the responsibility of the device-independent software to map the symbolic device names onto the appropriate device drivers. For instance, in Linux, devices appear as named objects in the file system. Each I/O device is uniquely identified by the combination of *major device number* and *minor device number*. The **major device number** is used to locate the driver associated with that device while the **minor device number** is used as a parameter to indicate the unit to be accessed.

An important issue related to device naming is the protection of I/O devices against uncontrolled access. The users should be able to access only those devices for which they have been permitted. For example, in UNIX, access to each device is controlled through `rwx` bits, which indicate the permissions for that device.

(b) Buffering

A buffer is a region of memory used for holding streams of data during data transfer between an application and a device or between two devices. Buffering serves the following purposes in a system :

- The speed of producer and consumer of data streams may differ. If the producer produces items at a faster speed than the consumer can consume or vice-versa, either the producer or consumer would, respectively, be in waiting state for most of the time. To compensate for this speed mismatch between producer and consumer, buffering may be used. Both producer and consumer share a common buffer. The producer produces an item, places it in the buffer and continues to produce the next item without having to wait for the consumer. Similarly, the consumer can consume the items without having to wait for the producer. However, due to fixed size of buffer, the producer and consumer still have to wait in case of full and empty buffer respectively. To resolve this, **double buffering** may be used, which allows sharing of two buffers between producer and consumer, thereby relaxing the timing requirements between them.
- The sender and receiver may have different data transfer sizes. Here again, buffers are used to cope with such disparities,. At the sender's side, large data is fragmented into small packets, which are then sent to the receiver. At the receiver's side, these packets are placed in a reassembly buffer to produce the source data.

(c) Error Handling

Many kinds of hardware and application errors may occur in the system during operation. For example, a device may stop working or some I/O transfer call may fail. The failures may be either due to transient reasons (such as overloaded network) or permanent (such as disk controller failure). The I/O subsystem protects against transient failures so that a system failure would not result. For instance, an I/O system call returns one bit that indicates the success or failure of the operation.

9.4.4 User-level I/O Software

While most of the I/O software lies within the operating system, a small part (user-level I/O software) contains the library procedures which involve I/O and run as part of user programs. These library procedures are linked with user programs during their execution. For example, if a user program contains a `read()` system call, then during execution the `read()` library procedure will be linked with the user program and included in the object code present in memory.

All user-level I/O software do not necessarily consist of library procedures; instead, some contain spooling system. **Spooling** refers to storing jobs in a buffer so that CPU can be utilized efficiently. Spooling is useful because devices access data at different rates. The buffer provides a waiting station where data can rest while the slower device catches up. The most common spooling application is print spooling. In print spooling, documents are loaded into a buffer, and then the printer pulls them off from the buffer at its own rate. Meanwhile, a user can perform other operations on the computer while the printing takes place in the background. Spooling also lets a user place a number of print jobs in a queue instead of waiting for each one to finish before specifying the next one. The operating system manages all requests to read or write data from hard disk through spooling.

LET US SUMMARIZE

- Controlling and managing the I/O devices and the I/O operations is one of the main responsibilities of an operating system. The operating system must issue commands to the devices to work, provide a device-independent interface between devices and the rest of the system, handle errors or exceptions, catch interrupts, and so on.