# Chapter 10
# Input-Output Management



Revised: Mar. 7, 2011

## 10.1 INTRODUCTION

The previous chapter has provided an introduction to input, output, and storage devices. As noted in that chapter, the operating system must provide I/O services to control all devices in the system; for example, to issue read/write commands, process device interrupts, and handle error detection and recovery for devices. The operating system must also keep track of processes waiting for the starting and completion of I/O events in order to schedule subsequent I/O operations, and make appropriate changes in process states that are caused by I/O events.

An OS component that manages a specific I/O device or type of device is called a **device driver**. A simple device driver for a clock was introduced in a previous chapter. Because of the wide variety of I/O devices, specialized device drivers are often needed for a particular installation; systems programmers are more likely to be called upon to write device drivers than any other type of OS component.

In a multiprogrammed or timesharing environment, there may be a high level of competition for the use of certain devices**. I/O scheduling** techniques may be required in order to control this competition.

As a preparation for learning to understand and write device drivers, it is necessary to have a basic understanding of the characteristics of the device to be controlled. Overviews of some common device types were presented in the previous chapter, along with a discussion of device interfaces. The hardware and system software must take a simplified view of I/O devices; this is discussed in Section 10.2. Section 10.3 deals with approaches to the implementation of device drivers. Once device and interface characteristics are understood, we can consider specific techniques for I/O programming. This is the subject of Section 10.4; further discussion about device management routines continues in Sections 10.5 through 10.8. Finally, when we have learned to manage individual devices, we are ready for the overall problems of I/O management and scheduling; this is the subject of Section 10.9.

## 10.2 The Processor's View of I/O

Every computer must provide instructions to control and access the I/O devices to which it is connected. Some older architectures offered an independent set of instructions tailored to each type of device that could be used. Today it is clear that the number of potential device types is far too vast to be supported on a case-by-case basis. Instead, generic instructions are provided, the meaning of which can be interpreted differently by each type of device.

With this approach, as discussed in the previous chapter, each distinct device is physically connected to a particular computer by means of an **I/O interface**. This interface allows the device to be accessed through a set of registers, which the CPU views as device registers. A simple device like a printer may have only one or two such registers, while a complex disk storage system may have dozens. Each device register is assigned a number to identify it to the system hardware and software and to provide an address for the I/O device. This number, called a channel number or port number, is established by the way the device is physically installed, and it can then be used by programs to refer to that device.

Because device registers have addresses, they can be considered to occupy an **address space**, just like memory locations. This address space, called the **I/O space**, is illustrated in Figure 10-1. In many computers, such as the Intel 80x86 processors, the I/O space is distinct from memory address spaces. It is accessed by a separate, usually small, set of I/O instructions that can read, write, or perform control operations on any specified device. Because these instructions are distinct, they may be designated as privileged instructions. This strategy provides the operating system with exclusive control of most I/O devices.

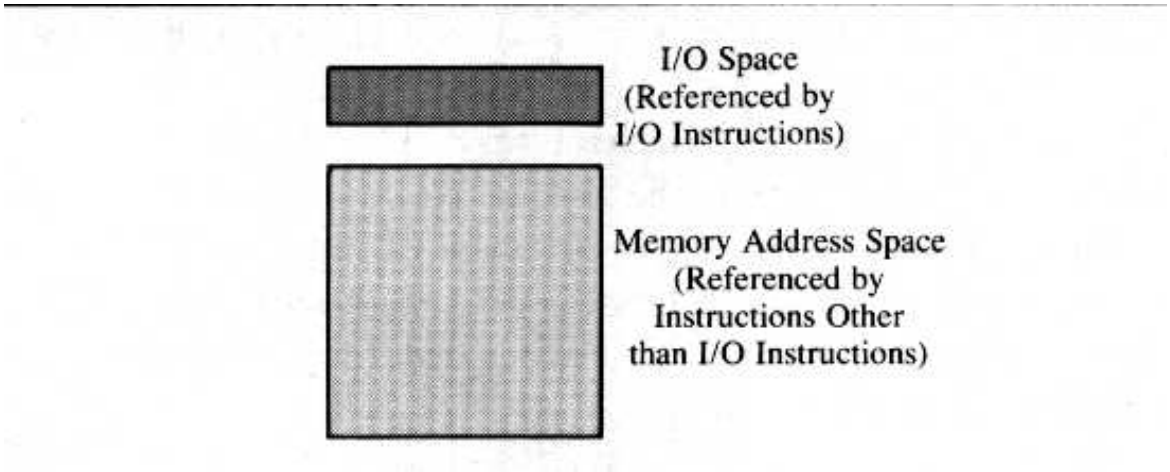


**Figure 10-1: I/O Space for Device Registers**

An alternate approach, used for example by VAX and Power PC computers, is to consider I/O devices and memory locations as sharing a single address space. This strategy, known as **memory-mapped I/O**, is illustrated in Figure 10-2. Because memory and devices are treated

uniformly, no distinct I/O instructions are needed. Instead, all instructions that read or write memory locations can perform the same operations on device registers just by using the appropriate addresses. In addition to simple data transfers, arithmetic and other data manipulations may be performed directly on the contents of device registers. A disadvantage of memory-mapped I/O lies in the fact that the devices cannot be protected by the privileged instruction mechanism. Instead, memory protection techniques must be used to limit the accessible regions of the address space.

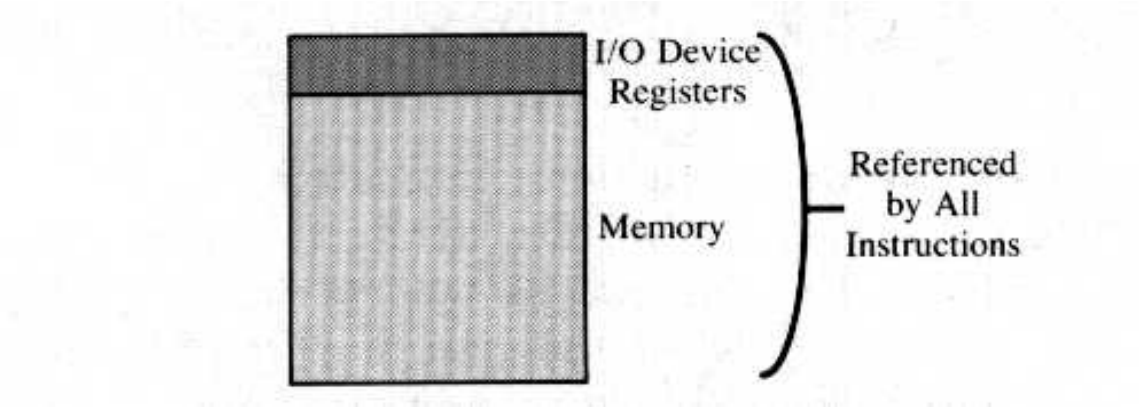


**Figure 10-2: Memory Mapped I/O**

## 10.3 DEVICE DRIVERS

In the majority of operating systems, all routines that access and control each type of I/O device are collected in a program unit called a device driver. Because of the great variation in the set of I/O devices used at each installation, almost all systems make some provision for adding new device drivers, and for adapting the set of drivers in use to the actual I/O configuration.

In some operating systems, device drivers are an integral part of the kernel, and can be modified only by a complete regeneration of the OS. In such systems, the I/O portion of the kernel is sometimes called the I/O supervisor. Adding devices to such an operating system cannot be done dynamically because the entire kernel must be regenerated to support a new device.

In most operating systems, though, the device drivers are structured as independent modules, stored on disk as files, and can be loaded when required by the operating system. This is especially beneficial for systems with limited main memory, since it allows loading of only the drivers required for devices that are actually in use. When the number of potential drivers is huge, this is the only approach possible. In these systems, a device table contains information about each device, including the name of the device driver to load. Adding a new device to such a system can often be done by a dynamic install command. This command checks for the presence of the device driver; if it is found, the device is added to the device table, provided that the table has a slot in which to store information about the new device.



-

Hence, there are generally two models for device support:

- device drivers are a part of the kernel of the operating system
- device drivers are separate modules that are dynamically installable and loadable

The first model is typified by early UNIX systems; the second by more recent OSs such as Linux or Windows XP. We shall use the term **I/O subsystem** to describe the I/O management portion of an operating system, regardless of which model of implementation applies. Figures 10-3 and 10-4 illustrate the two possibilities for implementing I/O subsystems.

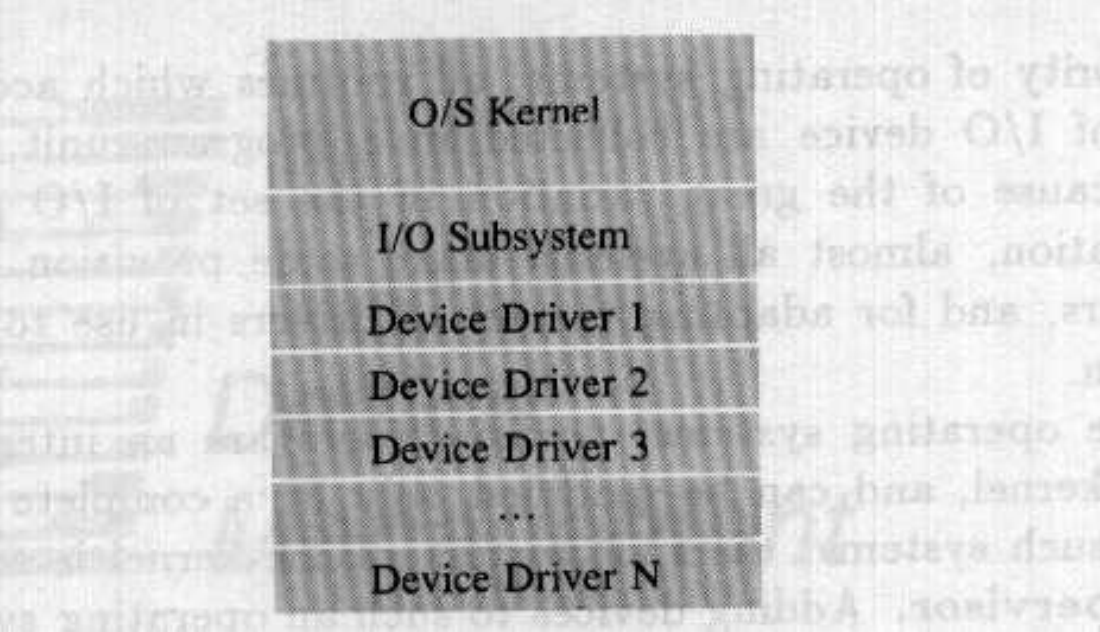


**Figure 10-3: I/O Subsystem as an Integral Part of the OS**

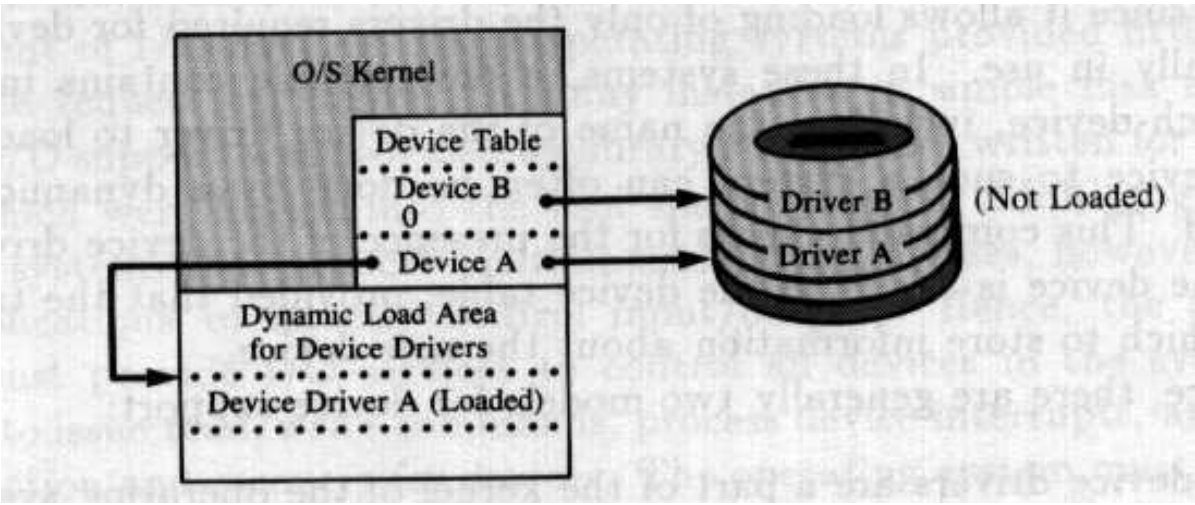


**Figure 10-4: Device Drivers as Separate Modules**

If reentrant programming techniques are used to implement device drivers in the I/O subsystem, each device type may require only one device driver. This means that all disk drives of the same type may be controlled by one driver, with different I/O control blocks used to represent the individual disk units. Similarly, a single device driver could control all terminals, another all printers, and so on. Of course, an installation may have several types of disk drives, for example, each requiring a separate driver. Generally, though, an operating system will have as few device drivers as necessary in the I/O subsystem to provide support for all devices actually present.

### *Device-independent Structure*

Since device drivers are frequently written by programmers other than the designers of the operating system, most OSs require device drivers to adhere to certain rules for implementation. Some systems try to maintain the concept of device-independent driver structure by which the structure of device drivers is more or less uniform across device types. Most UNIX device drivers, for example, are designed for two broad classes of devices: character-oriented (such as terminals and printers), and block-oriented (such as disks and tapes). Despite the wide variation in device details, all drivers within a class are expected to have the same overall structure. Furthermore, there is only one terminal driver, despite the wide variation in terminal types. This driver, however, has many options that can be set to match the characteristics of specific terminals.

From a user's point of view, device independence also means that a consistent set of operations, invoked by commands or by system calls, may be applied to most devices which logically could support them (the OS cannot make it meaningful to read from a printer!). The more independence of this type there is, the easier it will be for the user to utilize various devices in the system.

### *Preparation for Writing a Device Driver*

The implementation of device drivers is probably more common for systems programmers, and even application programmers, than the implementation of any other portion of an operating system, particularly in systems with dynamically loaded device drivers, in which adding device drivers is relatively straightforward. Because of this, design and programming guides for device drivers spelling out the necessary requirements are often included with the operating system documentation. These requirements must be followed if one expects to be successful in supporting such devices. Requirements or standards for implementing device drivers may include techniques for setting interrupt vectors, interface requirements, position independence, and setting device characteristics. Anyone who attempts to write a device driver must do some careful planning. You must keep in mind several important considerations in undertaking the implementation of a device driver:

- **Know the device**. Study the hardware manuals carefully to fully understand the characteristics of the device you are dealing with. Experiments with the hardware may be required to resolve ambiguities in the device manuals.



-

- **Know the structure of a device driver**. For any given operating system, consult the device driver information in the system manuals. Be sure to know every aspect of a device driver's structure.

- **Study other device drivers**. Many operating systems describe model device drivers in their documentation. Study the example(s) carefully. Use them as a guide for implementing the new device driver. Be sure to find a model that is similar to the device you are to support (e.g., a block device vs. a character-by-character device).

- **Consider special features of the device**. Most devices have unique features. Determine how these features affect the implementation and how you will support them.

A more detailed discussion of device drivers is given later in this chapter.

## 10.4 PROGRAMMING STRATEGIES

The general strategy for handling a particular device falls into one of three categories, depending on both the device type and the computer architecture:

- **Programmed I/O**. The programmed I/O strategy, also called simple I/O, is used on small and medium computers to manage character-by-character devices, which can produce or accept data one byte (or word) at a time, at the convenience of the CPU. A distinct I/O instruction is used to transfer every word of data to or from the device.

- **Block transfer**. The block transfer technique is used on small and medium computers for devices like disks and magnetic tapes, which must transfer data in large blocks at a high speed. The devices are attached to an I/O controller, which contains sufficient storage and control logic to independently manage the transfer of an entire block. The controller in turn is connected to the memory by a technique called direct memory access (DMA). The CPU sends the controller sufficient information to begin the transfer of the block. Then, the CPU can proceed to other work while the controller independently transfers the entire block to or from memory.

  For a disk, the information sent to the controller must include at least the following information:

  i) address of the block in memory

  ii) location of the block on the disk (track, sector, etc.)

  iii) size of the block

  iv) read or write request

- **I/O processors**. The I/O processors technique is used uniformly for control of most I/O devices on some larger computers, such as the IBM S/370 and its successors and the CDC Cyber series. Each device is connected to a special-purpose processor that is capable of fetching and executing instructions to direct an I/O transfer or a series of such transfers. Usually there are a small number of such processors, each in turn handling many devices. These processors may understand only a few special instructions, or they may have most of the features of a general-purpose CPU. Their programs may be stored in a private memory, or they may be obtained from the main memory of the computer system. This strategy allows the initiation of many different I/O requests by the CPU using one machine instruction. The processor then continues the I/O sequence by fetching and interpreting the instructions in the I/O program.

## 10.5 POLLING VS. INTERRUPTS

In this section we will examine two techniques for monitoring device activity or status: **polling** and the use of **interrupts**. As we shall see, polling can work for programmed I/O in some simple environments such as those of single user operating systems, but this is not generally acceptable in multiprogramming systems.

### *Polling*

The polling approach to device control requires that device status be checked by explicitly testing values in a status register. After a command is sent to a device, it must be polled to determine when the operation is completed so that the next operation, if any, can be begun. If the device is a character-by-character device, this sequence of I/O and polling must be continued until all characters have been transferred between the device and main memory. A simple polling algorithm for this type of I/O is given in Figure 10-5.

```
/* initialize pointer and index */
ix=0;
count = BUFLEN;

/* begin polling loop */
while ((count < 0) && (no device error)) {

/* wait while device is busy */
while (device is busy) {}

     /* process character if no error */
     if (no device error) {

          write buffer[ix]; /* write next character */
          count = count - 1; /* decrement count */
          ix = ix + 1; /* increment index */
     }
}
```

**Figure 10-5: A Simple Polling Algorithm for Device Control**

In this example, it is assumed that there are methods of testing the status of a device to detect busy and error conditions. In addition, a means of transferring the single character in the `BUFFER` array must be provided. In simple memory-mapped I/O, it could be possible to test the device registers themselves for various status bits (BUSY, ERROR) and to move the character directly into the memory location, which is the data transfer register for the device. The algorithm can then be modified to that shown in Figure 10-6.

```
/* initialize pointer and index */
ix=0;
count = BUFLEN;

/* begin polling loop */
while ((count < 0) && (error bit of status register is 0))
{

     /* wait while device is busy */
     while (status bit of status register is BUSY) {}

     /* process character if no error */
     if (error bit of status register is 0) {

          /* write next character */
          data register = buffer[ix];
          count = count - 1; /* decrement count */
          ix = ix + 1; /* increment index */
     }
}
```

**Figure 10-6: Simple Polling with Memory-Mapped I/O**

The algorithms depicted in Figures 10-5 and 10-6 are typical for polling control of a device. No device executes an I/O operation if its status indicates that it is busy or an error condition exists. It is possible to have a polling type of operation on a DMA device as well, although the CPU will not have to be involved in the data transfer itself once the I/O command has been sent to the device or device control unit.

As an illustration, Figure 10-7 shows the use of polling for a simple diskette device on an early minicomputer, which requires a sequence of information to be sent to the device before it can respond to the requested command. A read operation on this diskette unit consists of two I/O operations: read sector (which transfers data from the diskette to an internal buffer in the control unit) and empty buffer (which uses DMA to transfer data from the internal buffer of the control unit to the internal memory of the computer). A protocol consisting of device communication to the CPU, followed by the CPU's response, is required after each operation.

```
move read-sector command to control status register;
while (transfer bit of control status register is OFF) {}
move sector address to data register;
while (transfer bit of control status register is OFF) {}
move track address to data register;
while (done bit of control status register is OFF) {}
move empty-buffer command to control status register;
while (transfer bit of control status register is OFF) {}
move word count to data register;
while (transfer bit of control status register is OFF) {}
move memory buffer address to data register;
while (done bit of control status register is OFF) {}
```

**Figure 10-7: Polling a Simple DMA Device**

Upon receiving the read sector command, the device sends back a transfer request signal to the CPU via the device status register. The CPU responds by sending the sector address to a data register. The device once again responds with a transfer request to the CPU, and the CPU responds with the track address. At this point in time, the read sector operation is active. The device status will indicate BUSY, or not DONE, so the program can now poll the status register for the DONE state. Once DONE is indicated, an empty buffer command is sent to the control unit, which once again responds with two transfer requests: one for the number of words to transfer, and the other for the address in main memory that will be used to store the data. Once again, the status register is tested for DONE while the data transfer to memory is taking place. Note that polling actually takes place at many points during the operation, both to check for readiness after transfer of each item of control information, and to wait for each of the two operations to be complete.

In all of the previous examples, we assumed that no interrupts were generated by the device. Systems capable of such polling in general have a means of disabling any interrupts that might occur. The simplest device driver might be little more than an extension of these simple polling models. This would be a possibility, but the obvious drawback is that the CPU is going to waste much time in polling loops waiting for devices. More importantly, the basic principle of multiprogramming systems, that one program or process may execute while another is blocked waiting for I/O completion, would be violated. Hence, device drivers implemented with polling cannot be used in a multiprogramming environment.

### *Use of Interrupts*

The use of interrupts in device management allows an I/O operation for a device to be started by the device driver, and the device driver to return to the process that made the I/O request (or, as we shall see later, pass control to the operating system to schedule another process if the requesting process is blocked waiting for I/O completion). In this case, a device interrupt handler becomes an integral part of the device driver. This interrupt handler receives control when the device (or control unit, or channel) generates the interrupt signal, which causes the CPU to transfer to the handler address. Obviously, how this is done in the hardware is dependent on the architecture of a particular computer system.

In most operating systems some type of data structure, either a simple one-bit flag called an event flag, or a more elaborate structure called an event control block (ECB), is used by a process or the operating system to represent events, such as the completion of a particular I/O operation. When simple I/O is performed with interrupts, although characters are actually transferred one at a time, use of event flags or ECBs can make the device appear to operate the same as a DMA device. This is because the signaling of an I/O event's completion is done by this event flag for both types of data transfer.

To illustrate this process, Figure 10-8 presents a modification of the polling algorithm from Figure 10-5. In this case, device initialization involves setting up a count and a buffer pointer, and enabling interrupts for the device, which will be processed by the device interrupt handler when they occur. As each character is transferred, the interrupt handler receives control, processes the character, and---when all characters have been written---sets a device **event flag** to indicate that the requested operation is complete. Although this event flag is polled by a simple loop in the example, it could be tested at any point in a program without affecting the progress of data transfer to the device.

In a single-process environment, the interrupt handler in Figure 10-8 would simply return to the interrupted program, at the point of interruption, and resume execution. In a multiprogramming system, interrupts provide an opportunity for forcing a context switch when a device completes its requested operation. The device event flag could be a part of the PCB itself, and could be used to determine if a process is to change from a blocked state, waiting for device I/O, to the ready state. In this case, control may not return to the interrupted program. Instead, control may transfer to the operating system, which ultimately invokes a dispatcher to schedule the next process to use the CPU. This would be a likely outcome when preemptive scheduling is used, whereby low-priority programs lose control of the CPU if a higher-priority process becomes ready. Since the completion of an I/O operation requested by a process may indeed cause a process to become ready, the operating system must be able to get control for a possible context switch.

```
MAIN PROGRAM:

/* initialize index, count, and device flag */
ix = 0;
count = BUFLEN;
dev_flag = 0;
...
enable interrupts for device
...
/* loop until all characters are written */
while (dev_flag == 0) {}
...

INTERRUPT HANDLER:

save registers

/* check for errors */
if (error) {

     dev_flag = -1; /* set flag to error code */
     disable interrupts for device
     restore registers
     return from interrupt
}

/* check for completion */
if (count==0) { /* operation completed */

     dev_flag = 1; /* set flag to completion code */
     disable interrupts for dvice
     restore registers
     return from interrupt

}

/* no error, not complete; process character */
write buffer[ix]; /* write next character */
ix = ix + 1; /* increment index */
count = count - 1; /* decrement count */
restore registers
return from interrupt
```

**Figure 10-8: Use of Interrupts and Event Flags for Simple I/O**

## 10.6 BUFFER MANAGEMENT

The time required for physical I/O on devices is quite long when one considers the number of instructions that a CPU can execute during this time. One goal in an operating system would be to minimize device idle time, hence maximizing device throughput. We can use a technique called **buffering** to improve device throughput as well as CPU throughput. Buffering, in this context, is the use of temporary storage areas in memory (buffers) to store data that is read from an input device before it is needed or can be used by a process, or to store data produced by a process before it can be accepted by an output device. We will discuss three types of buffering.

### *Ring Buffers*

Character input devices can often have a complete block of data received and stored (buffered) by a device driver before this block is transferred to a process requesting it. (see Figure 10-9). Often, character buffering for input in such a device driver uses a FIFO buffer, called a **ring buffer** because of the way it operates. The buffer is processed in a ring fashion, with an ordering such that the next input data character after the character stored in the last physical location of the buffer is stored back at the beginning of the data area (if it is free); in effect, characters in the buffer form a circular list. Two pointers, `RING_IN` and `RING_OUT`, manage the buffer. `RING_IN` keeps track of the next free location into which the interrupt handler will place the next character read. `RING_OUT` determines the next character to be placed in the process buffer. A ring buffer counter and the ring size are maintained so that we know when the ring buffer is full or empty. The `RING_IN` and `RING_OUT` pointers can never "pass each other." `RING_IN` is manipulated by the device driver interrupt handler, while `RING_OUT` is manipulated by the device driver read routines, which send data to the process buffer. Figure 10-10 illustrates the ring buffer concept.

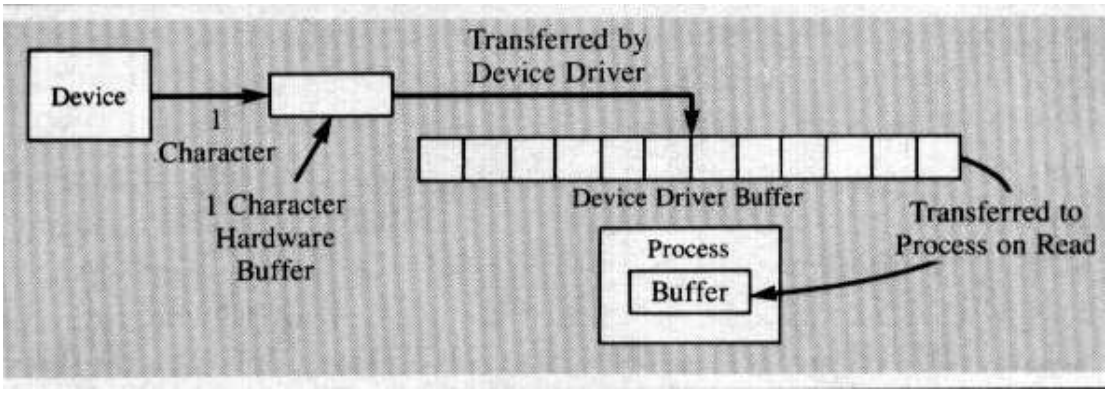


**Figure 10-9: Buffering for a Character-by-Character Device**

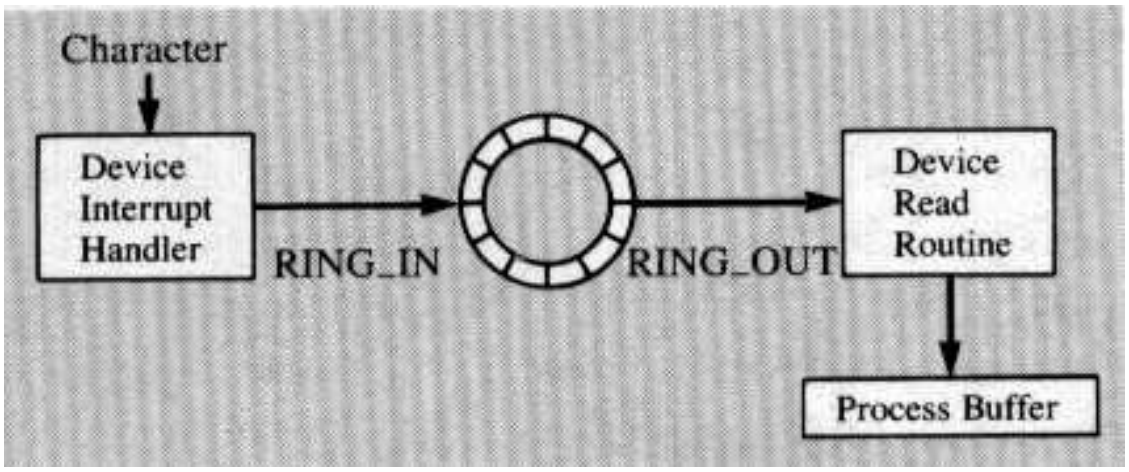


**Figure 10-10: Use of a Ring Buffer for Buffering Input**

Device drivers for DMA devices can also buffer data on block transfers. The most typical buffering occurs when data are blocked (multiple logical records in one physical record), and only one logical record is transferred to a buffer of a requesting process each time a read is requested. This buffering is most effective in sequential operation, whereby logical records will be requested in the order in which they appear in the physical block. A physical block with seven logical records would have seven logical records buffered, and would only do a physical read on every seventh read request, as illustrated in Figure 10-11.

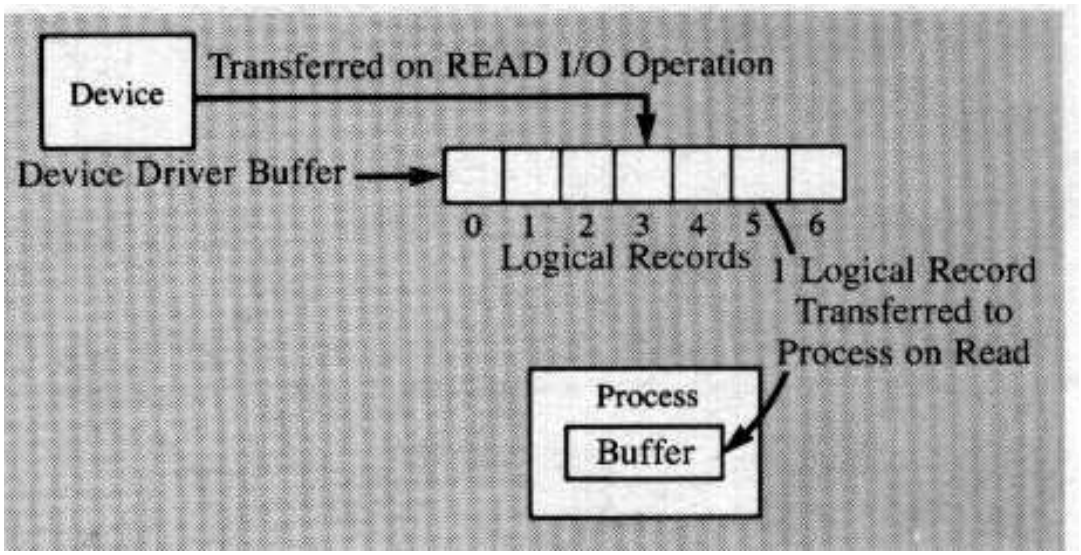


**Figure 10-11: Buffering for a DMA Device**

Similar buffering to build physical blocks occurs on the output side prior to the writing of data. In the case of output data, it is easy to see why multiple records could be lost if the system fails, requiring a reload of the operating system.

### *Double Buffering*

The device management routines of an operating system should provide features that both maximize the throughput of individual devices, and support the operation of devices simultaneously with other devices or CPU activities. The buffering techniques just described do help, but additional techniques can provide further improvement. **Double buffering** is a technique for gaining speed in a data transfer by the use of two or more buffers by a device driver, or perhaps by a process. While one buffer is being processed by the CPU, another buffer may be read or written by the device. In a device driver supporting a block device with multiple logical records, we can use double buffering to improve performance in at least two ways:

1. Upon request for the last logical record in the internal buffer of a device driver, initiate the read of the next physical record, even though it is not known that this physical record will be required. This allows the device read to proceed while a record is being processed, resulting in overlapped I/O and processing.

2. Provide two buffers for physical records. Upon opening the device, two physical records can be read "immediately," one after another. Then, when records are exhausted in buffer one, a read can be initiated to read more data into buffer one, while records from buffer two are available immediately.

The use of two buffers can be very significant when blocking is small or non-existent because the delays waiting for the next physical read can be greatly reduced.

It is possible to allow an application process to manage buffers of this type, and gain increased throughput, if the operating system supports asynchronous I/O, that is, I/O operations such as read and write which return control to the caller as soon as the operation has been set up, even though the actual transfer has not completed or perhaps has not even begun. This means that I/O system calls do not automatically result in the process being blocked, as illustrated in Figure 10-12. While this capability is of little value in many high-level programs, which typically assume that read or write operations will be completed before any following statements will be executed, it is of great use in other applications, especially real-time processes and many system programs. The requesting process must be provided with an event flag or an additional system call, which can be used to determine if the previous I/O operation has completed.

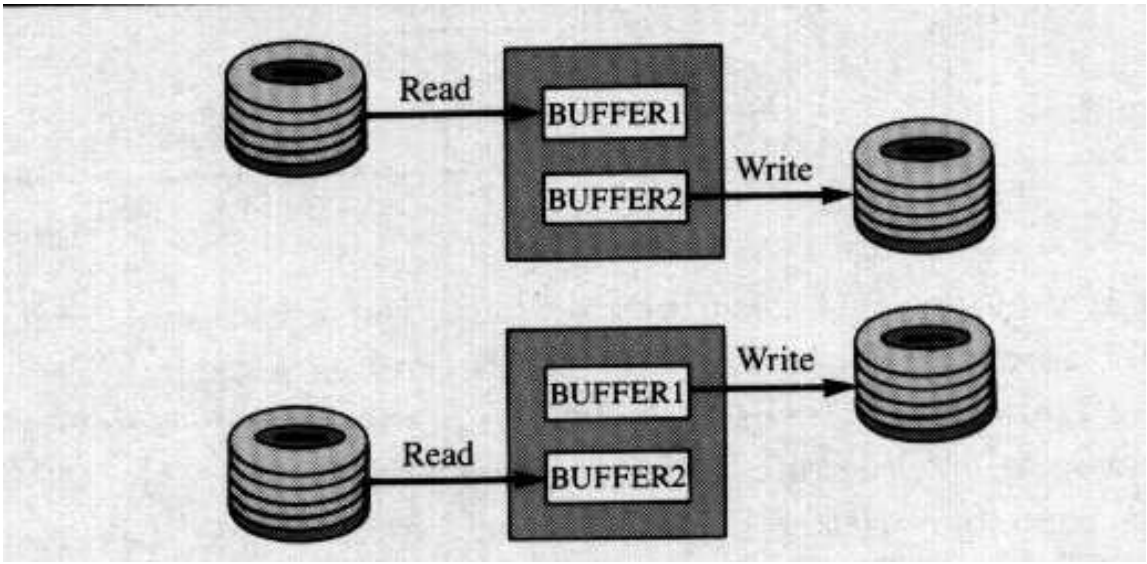


**Figure 10-12: Overlapped I/O on Two Devices**

The advantages of double buffering by a process may be illustrated by a simple copy program. Suppose such a program is to copy a file from a tape on tape drive 1 (TAPE1) to tape drive 2 (TAPE2), each tape drive having its own controller so that tape I/O operations can overlap. If we use the typical READ/WAIT, WRITE/WAIT sequence, we might follow this procedure:

```
WHILE NOT END_OF_FILE(TAPE1)
   READ TAPE1,BUFFER
   WAIT TAPE1
   IF NOT END_OF_FILE(TAPE1)
      WRITE TAPE2,BUFFER
      WAIT TAPE2
   ENDIF
ENDWHILE
```

The device throughput for both tapes would be approximately fifty percent of their maximum rates because while TAPE1 is busy, TAPE2 is idle, and vice versa.

Now consider the following double buffering algorithm:

```
READ TAPE1,BUFFER1
WHILE NOT END_OF_FILE(TAPE1)
   WAIT TAPE1
   WAIT TAPE2
   IF NOT END_OF_FILE(TAPE1)
      READ  TAPE1,BUFFER2
      WRITE TAPE2,BUFFER1
```

```
      ENDIF
      WAIT TAPE1
      WAIT TAPE2
      IF NOT END_OF_FILE(TAPE1)
         READ  TAPE1,BUFFER1
         WRITE TAPE2,BUFFER2
      ENDIF
   ENDWHILE
```

Here it is assumed that `READ` and `WRITE` pass control back to the program upon initiating the I/O request. The program must check the I/O operation with the `WAIT` operation. If both tape drives operate at the same speed, near-maximum throughput will be realized (OS overhead or other use of the tape controllers may prevent throughput from reaching 100%). Now, while `TAPE1` is reading, `TAPE2` is writing, and vice versa. Note the beginning read of `TAPE1` was required to get the double buffering scheme "rolling."

If the devices used do not operate at the same speed, double buffering still helps, but improvement is limited to the amount of time required for the faster device's I/O operations, as illustrated in Figure 10-13.

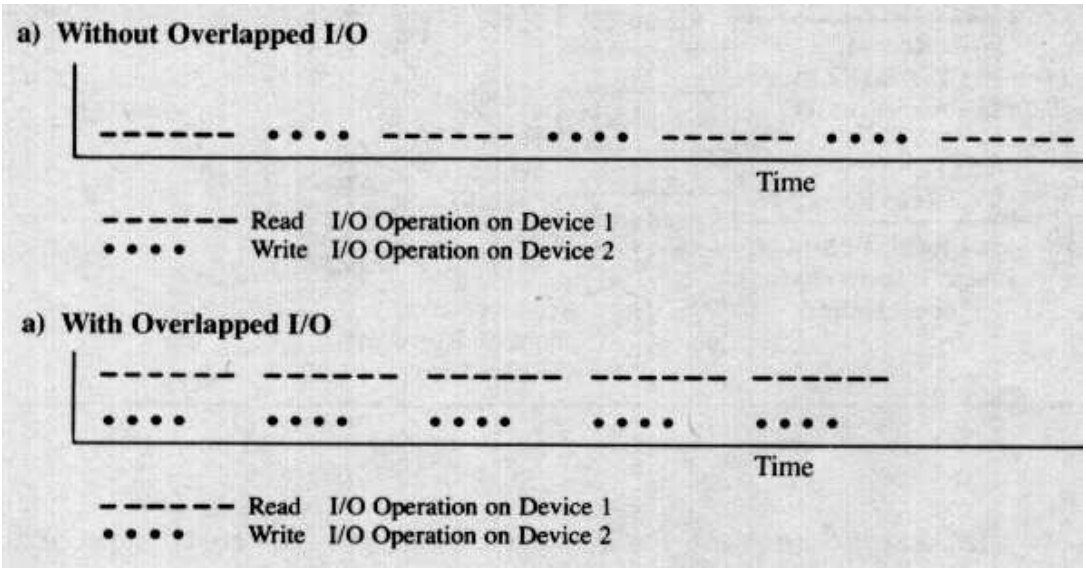


**Figure 10-13: Improvement in Throughput using Double Buffering**

### *Software Caching*

We can improve device performance by eliminating a certain percentage of reads and writes for a device by a different buffering technique called **software caching**, similar to hardware caching. This approach is now commonly used in disk file servers in local area networks to improve performance.

Let us consider an environment supporting disk caching. A set (cache) of buffers is allocated to hold disk records that have been recently read or written. The size of each buffer is usually a multiple of the sector size on the disk being supported. Consider the situation where five "records" (identified by a number for the sake of this example) have been read in an environment with five disk cache buffers in the computer's memory, as shown in Figure 10-14. Now suppose a write of record 72 is requested by a process. In this environment, no I/O takes place (at least not immediately); rather, the disk caching facility simply updates the record in memory, thus avoiding an I/O operation. If a read of record 104 is requested, the record is provided from the memory-resident disk cache buffer, and no I/O takes place. The assumption is that recently accessed records will tend to be accessed again, and thus should be kept in the cache; a similar assumption underlies the operation of virtual memory systems, to be discussed in a later chapter.

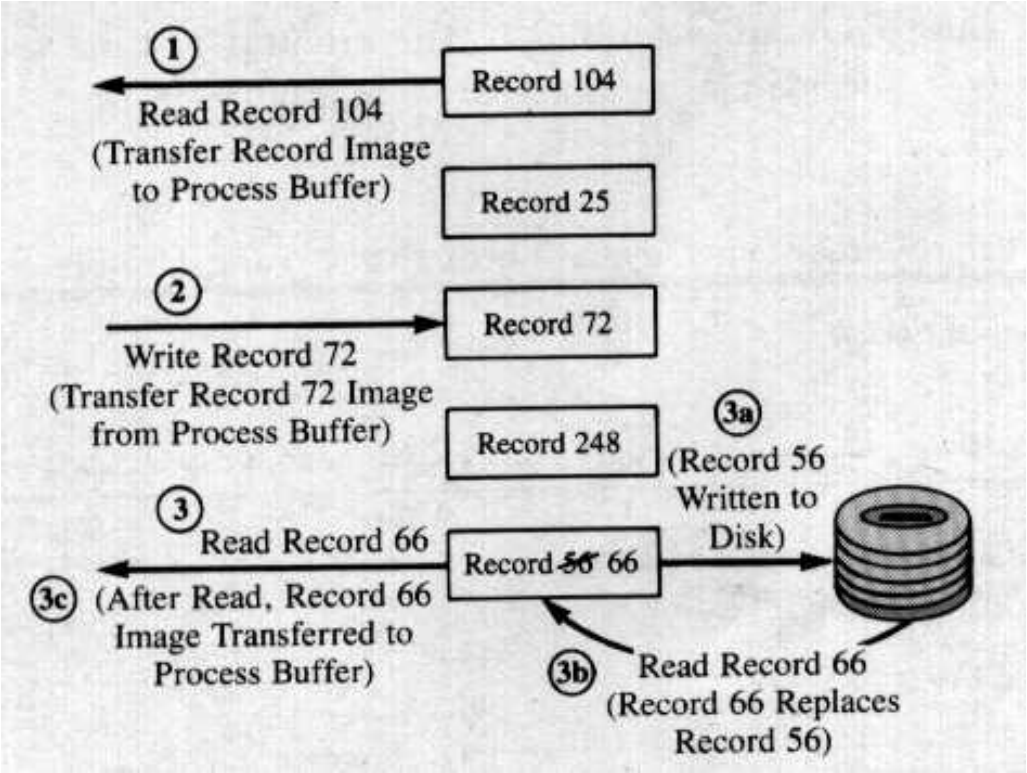


**Figure 10-14: Use of Disk Cache Buffers to Improve Performance**

Suppose now that a read request for record 66 occurs. Since this record is not in the cache, a cache buffer must be freed by writing the contents of the buffer to disk (if it is not the same as the record on disk, i.e., it has been changed by a previous write operation). The cache software algorithm selects a buffer to be used for the read and, if necessary, writes its contents to disk before reading record 66 into this cache buffer. The cache buffer chosen usually reflects recent usage of disk cache buffers; for example, it selects the least recently used buffer.

Disk throughput improves in this instance because many disk I/O operations are eliminated for repeatedly used records that remain resident in disk cache buffers. An environment such as

one with a repeatedly used database index would be a likely one in which improvements in disk performance could be easily observed because the index, or portions of the index, would remain resident in the disk cache. Note that records referred to in this example are actually areas of the disk (contiguous sectors) and many logical records might be contained in a single cache buffer.

A disadvantage of software caching is that cache buffers could be lost in a system failure, resulting in a corrupted file system or files. Systems supporting disk caching usually provide for the automatic, periodic writing of the disk cache buffers at intervals specified by a system operator, and for the writing of disk cache buffers via a system command or on system shutdown. In such environments, one does not simply turn off the computer---it must be shut down in an orderly manner. UNIX provides disk caching, which is the primary reason that orderly system shutdowns are required before powering off the computer system. To minimize the problem, UNIX provides a system call and command (sync) to flush all buffers to the disk, and automatically executes this command every few minutes.

## 10.7 DATA STRUCTURES FOR DEVICE MANAGEMENT

I/O operations that are in progress or have been requested are represented by some type of data structure in every operating system. The format for such data structures is determined by the rules for device driver implementation. This data structure is called by various names, including device control block (DCB), I/O block (IOB), I/O control block (IOCB), unit control block (UCB), and channel control block (CCB). It is a data structure that represents a device, channel or controller to the device driver, and an activity of the device, channel or controller to the operating system and/or a process. Some operating systems have a hierarchy of these control blocks, all requiring access by the device driver. For example, in the VAX/VMS operating system, a channel control block points to a unit control block, which in turn points to other control blocks. Hence, data structures for representing device, controller, and channel activity can become quite complex. In this text, we use the term I/O control block, or IOCB, for a data structure representing I/O activity. Figure 10-15 lists some information that might be found in a typical I/O control block.

---

```
Channel Number
Controller Address
Device Name
Device Address
Interrupt Vector Address
Address of Interupt Handler
Device Type
Address of Open Procedure
Address of Close Procedure
Address of Start I/O Procedure
Address of Cancel I/O Procedure
Buffer Address
```

```
Buffer Length
Current Buffer Pointer
Current Data Count
Current I/O Operation
Address of PCB of Process which requested the Operation
Address of I/O Request Parameters
Address of ECB for Current Operation
```

**Figure 10-15: Information Stored in an I/O Control Block**

In addition to IOCBs, device tables representing the various device drivers in the operating system must be maintained. Operating systems that allow dynamic installation of device drivers would perform the installation by adding entries to such a device table. The contents of a typical device table entry are shown in Figure 10-16.

```
Device Name
DeviceStatus
Device Driver File Name or Disk Address
Entry Point in Main Memory, if loaded
Device Size
Device Driver Size
Logical Name(s)
```

**Figure 10-16: Content of a Typical Device Table Entry**

Once again the complexity of implementing device drivers increases, since these additional data structures must be used and understood in a given operating system environment.

## 10.8 DEVICE DRIVER ORGANIZATION

Device drivers for any real general purpose OS are enormously complex. A book would be required to fully describe them. The suggested readings at the end of the chapter contain references for several such books. We will concentrate only on the most fundamental requirements.

Every device driver must provide mechanisms to support certain basic activities for each device. These activities include:

- preparing for I/O
- starting I/O
- interrupt servicing
- error detection and recovery
- completion of I/O
- cancellation of I/O

In addition to these activities, an I/O system call interface is needed, including procedures to service common system calls; the operations supported generally must include open, close, read, write, and a variety of control operations. These system calls, when directed to a specific device, invoke components of the corresponding device driver in executing each specific operation.

Most device drivers and control routines are divided into two parts: the main driver and the interrupt handler. Figure 10-17 illustrates a common structure along with the functions provided by each component.

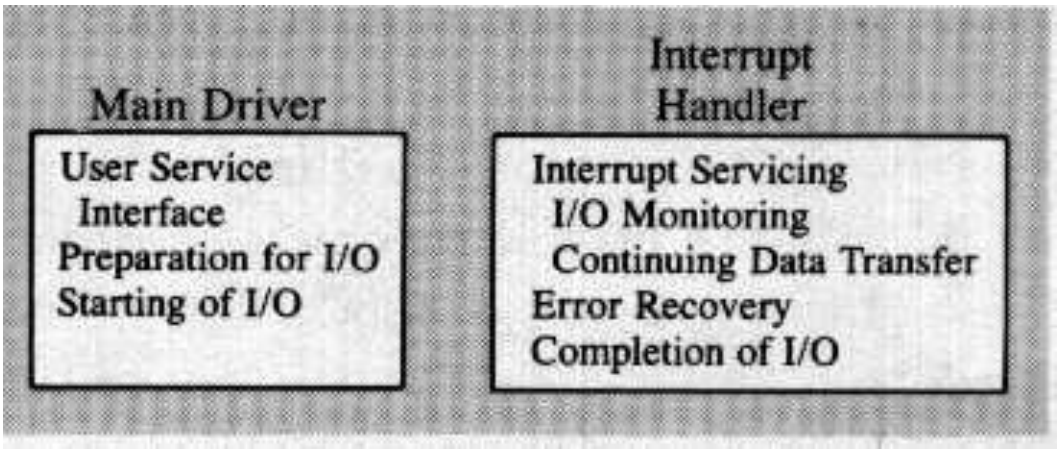


**Figure 10-17: Structure and Functions of Device Drivers**

The two-part structure of such a device driver is such that the main driver receives control to check the validity of the I/O request, prepare for that I/O request, and start the I/O. Rather than polling for I/O to complete, the driver then returns control to the caller, and the interrupt handler intermittently receives control until the I/O operation is completed. This process is illustrated in Figure 10-18.

### *I/O System Call Interface*

Operating system standards for system calls and device driver implementation specify how a device driver gets control and how parameters are passed to the device driver. System call standards depend on both the operating system and the architecture of the computer system. In some systems, different instructions can generate a supervisor call or trap; in others only one instruction type serves both purposes..

Once the device driver gets control, what is the format of the parameters? How are they found? Most systems will specify a parameter list with a register pointing to the address of this list. All of this is determined by the I/O system call interface standards.

### *Preparing for I/O*

Preparation for I/O is the first activity of a device driver when invoked via one of the device system calls. Since device drivers must adhere to operating system interface standards,

the I/O system call parameters must specify the type of operation requested (read, write, control), the address of the data buffer to be used, the size of the data area, and other relevant information. In addition, device-dependent parameters, such as track and sector numbers for disk operations, may also be required.

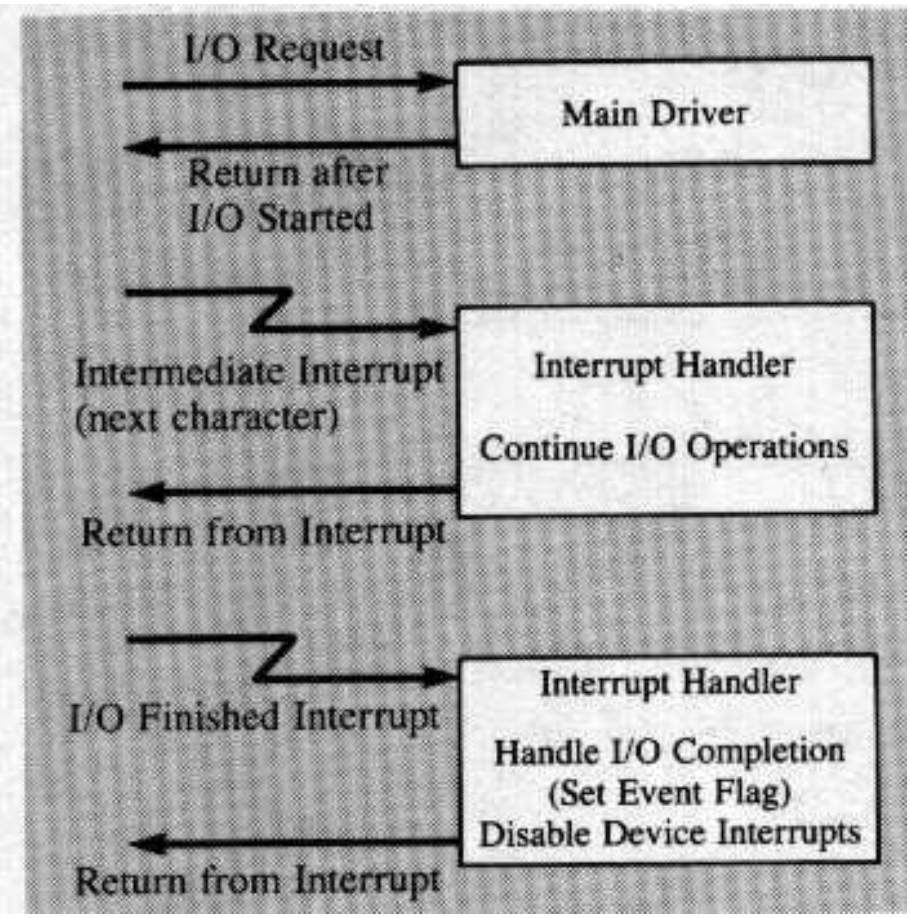


**Figure 10-18: Flow of Control in a Device Driver**

Preparation of I/O components of a device driver validates device-specific parameters of an I/O request. Appropriate error return codes must be provided to the user for any errors detected in the service request parameters.

The preparation for I/O might include temporary buffer allocation or initialization, formatting of data, and placing information in the appropriate location(s) in some I/O control block that is accessible to the device interrupt handler and the operating system. Once the I/O request parameters are validated, the device status must be checked. If the device is busy or not ready, the device driver must take appropriate action. This action is dependent, of course, on the operating system. If an error condition or problem such as "device not ready" is detected, the

driver might take action causing the application process to terminate, or it might simply provide a return code to be passed back to the process, which then must deal with the problem in a suitable way depending on the application. In a multiprogramming operating system, the response may include queueing the I/O request for a shared device that is presently busy with another operation.

### *Starting I/O*

If the requested device is not busy and no abnormal conditions were detected, the driver starts the actual I/O operation. Once again, the means of doing this is both device-dependent and computer-dependent. In some instances, the driver simply enables interrupts for a device. In that case, all other device service is provided by the interrupt handler.

Some computers have specific I/O machine instructions to begin operations, such as eXecute I/O (XIO) or Start I/O (SIO). Computers using memory-mapped I/O start I/O by writing to device control registers using addresses in the memory space. Figure 10-19 lists the type of information required to start I/O operations for various devices. This information may be provided via an operand to the I/O instruction or by special operations after the I/O instruction is issued.

| DEVICE | REQUIRED I/O INFORMATION |
|---|---|
| Disk | Operation Code<br>Memory Transfer Address<br>Number of Bytes to Transfer<br>Track Address<br>Sector Address |
| Tape | Operation Code<br>Memory Transfer Address<br>Number of Bytes to Transfer |
| Printer<br>Terminal<br>Serial Device | Operation Code (read/write)<br>Character to Transfer |
| Timer | Time interval |

**Figure 10-19: Information Required by Devices when Starting I/O**

Starting I/O can be as simple as enabling interrupts so that the interrupt handler can continue to monitor I/O operations. This is particularly true of character devices. In more complex devices, an I/O command may have to be built and sent to the device with the appropriate I/O instruction. Once I/O is started, control is returned either to the caller or the device driver, which may in fact be a portion of the operating system.

### *Interrupt Handling*

The most complicated part of most I/O drivers or I/O supervisor modules is the interrupt handler. Much of the device control is embedded in the interrupt handler, which is given control asynchronously when the device needs service. This could involve processing the next character for a device in the case of a non-DMA device, or starting a second or third phase of an I/O operation for DMA devices (for example, a read after a seek to sector on some disks).

Information needed by the interrupt handler must be provided by the main portion of the device driver. Such information is usually contained in the I/O control blocks that represent the current I/O operation, just as PCBs represent process execution to the dispatcher.

Device interrupt handlers must save all registers and hardware status of the interrupted program. In a stack machine, this merely involves saving information on the stack and restoring it before returning to the interrupted program. Recall that the program counter and program status word are automatically saved by the hardware interrupt.

Certainly the most complex part of interrupt handler service is error recovery (which we'll cover in the next section). Error recovery routines can be either resident or dynamically loaded when needed, depending on operating system design and requirements.

### *Error Detection and Recovery*

Implementation of device drivers would be far simpler if one could assume an error-free environment. The same is true for all programs. Unfortunately this is never the case. Just what types of errors can occur? The answer, of course, depends on the type of device. A summary of some common errors for various device types is presented in Figure 10-20.

| DEVICE | POSSIBLE ERRORS |
|---|---|
| Disk | Invalid Track, Sector<br>Wrong Density<br>Power Unsafe<br>Data error |
| Tape | Data error<br>End of tape |
| Printer | Paper out<br>Paper jam<br>Off line |

**Figure 10-20: Possible Errors for Various Devices**

Just as the types of errors vary, appropriate recovery methods also vary from device to device. Recovery may be as simple as reporting a printer out of paper to the user or, more likely, to the operating system for operator intervention and subsequent continuing of the I/O operation. In other cases, the error recovery may be quite complex. As an example, the algorithm in Figure 10-21 illustrates an error recovery procedure for a magnetic tape drive, which is invoked when the device signals an error during an attempt to read a block of data.

```
ERROR HANDLER:

/* initialize counter and flag */
loop = 0;
tape_error = TRUE;

/* repeat entire process up to ten times */
while ((loop < 10) && tape_error) {

     /* backspace and reread up to nine times */
     count = 0;
     while ((count < 9) && tape_error) {

          backspace record
          read record again
          if (read is ok) tape_error = FALSE;
          else count = count + 1;
     }

     /* if error still present, backup further */
     if (tape_error) {

          /* backspace ten records */
          count = 0;
          while (count < 10) {

               backspace record
               count = count + 1;
          }

          /* skip forward nine records */
          count = 0;
          while (count < 9) {

               forward space record
               count = count + 1;
          }

          /* try reading again */
          read record
          if (read is ok) tape_error = FALSE;

          loop = loop + 1; /* prepare for retry */

     } /* repeat if necessary */
}
```

**Figure 10-21: An Error Recovery Algorithm for a Tape Drive**

The algorithm shown in the figure retries the tape read nine times by backspacing a record and then rereading. If the error persists, the tape is backspaced ten records, forwardspaced nine, and then reread (to try to "clean off" the tape). This entire procedure is repeated ten times, yielding a total of 100 retries. The actual implementation of this process involves starting I/O operations in the interrupt handler, then servicing the interrupts for these error recovery operations when the interrupts occur. Thus, the interrupt handler must know it is performing error recovery and keep track of where it is in the overall recovery procedure.

This algorithm seems complex enough, but consider the case of an error occurring during the error recovery procedure. Suppose that the record in error is only the fourth record on the tape. Then, when the fifth backspace operation is attempted while trying to backspace ten records, a "backwards at load point" signal will occur, indicating an attempt to backspace past the beginning. While not really an error, this is an unexpected condition and must be provided for in the logic of the interrupt error recovery routine. Note that the error recovery algorithm in Figure 10-21 does not handle this unexpected condition.

In spite of the complexity, the device driver must do all that is possible to recover from errors, particularly read or write errors to magnetic media. If recovery fails, it is important to preserve the integrity of the data as much as possible. Users may wish to operate as best as possible on erroneous data, but the indication of such non-recoverable errors must be provided to the application program so that it may decide how to proceed in light of the error.

### *I/O Completion Processing*

Once the requested I/O operation has finished, certain cleanup functions remain. These include setting the status of the process requesting the operation, clearing the device of busy status, and possibly disabling interrupts. One other function might be to search a device queue for another operation pending for the device that is now free, thus maximizing device throughput. I/O completion processing is usually a subcomponent of the device interrupt handler.

The responsibilities of each component of the device driver are summarized in Figure 10-22.

---

```
I/O System Call Interface

     Mechanism to invoke device driver
     Format of parameter list
     Locating parameter list

Preparation for I/O

     Validate parameters
     Save parameters in I/O control block
     Pass error codes to process or to OS
     Place in I/O queue or invoke start component
```

```
Starting of I/O

     Enable Interrupts
     Issue first I/O command to device
     Return to calling process or to OS

Interrupt Servicing

     Save registers
     Determine cause of interrupt
     If error, invoke error recovery routine
     If operation not done, issue next I/O command

          advance pointer(s)
          decrement count(s)
          restore registers
          return from interrupt

     If operation complete, set event flag

          invoke completion routine

Error Detection and Recovery

     Invoke error recovery algorithm
     If error recovers, resume interrupt handler
     If error is permanent, set event flag to error code

          invoke completion routine

Completion of I/O (Return to Idle)

     Set event flag
     Disable interrupts
     Move process just finished with device to ReadyQ
     Invoke I/O scheduler
     Invoke dispatcher

Cancellation of I/O

     If I/O request in I/O queue, remove it
     Return to caller
```

**Figure 10-22: Device Driver Components and Responsibilities**

## 10.9 I/O SCHEDULING

Scheduling I/O for a device can range from a very simple problem to one that is quite complex. In a single-user operating system, I/O requests can be serviced by suspending the process requesting the I/O and waiting for the I/O to complete. As we discussed previously, I/O requests should allow overlapped I/O and execution of the calling program if the calling program indicates it wants control back prior to the completion of the I/O. In such cases, either a subsequent system call (e.g., `checkio` ) or an event flag or ECB (as described previously) can be used to determine the status of the I/O operation.

### *Multiprogramming Environments*

Multiprogramming systems require that device queues be supported. Such device queues are needed to service I/O requests for devices that are not ready (tape or disk not mounted or printer out of paper) or that are shared devices or control units busy with another process's I/O request. Support of such queues includes routines to add requests to a device queue and to delete requests (should an I/O request be terminated before it starts). Another requirement in such a system will be to respond to I/O completion of one request with a search of a device queue to start the next request. This requires that the interrupt handler for a device be able to invoke the device scheduler at the completion of I/O for a device.

*I/O Queues*. The I/O queues that represent pending I/O requests have two possible structures:

1. a single I/O queue for all devices
2. multiple I/O queues (e.g., one for each device and/or controller in the system)

A queue entry in such queues would consist of a PCB address of a process with an I/O request waiting to be started. Figures 10-23 and 10-24 illustrate the two queueing possibilities. There are various ways of representing pending I/O requests and active I/O for a process. We describe here just one possibility, involving two distinct, mutually exclusive *blocked* states**, IO_init** and **IO_active.**

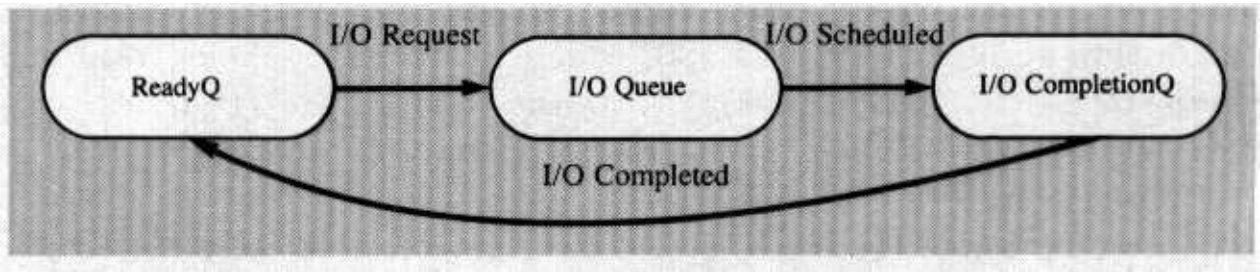


**Figure 10-23: A Single I/O Queue**

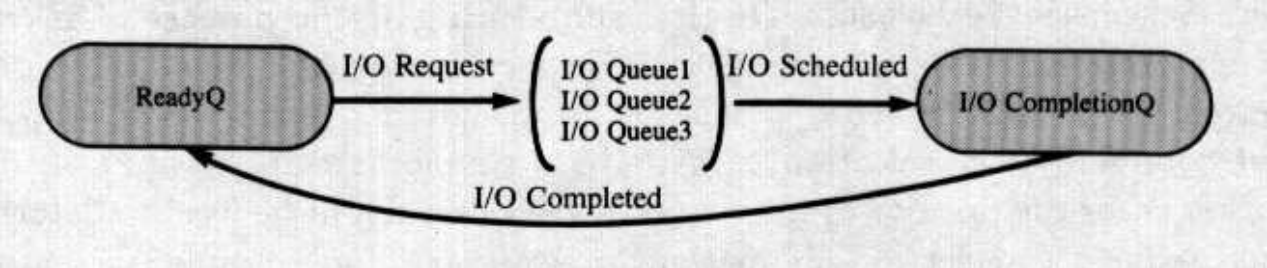


**Figure 10-24: Multiple I/O Queues**

If a valid I/O request is received but the device is not presently available, the requesting process is placed in the blocked state *IO_init*, indicating it is waiting for the initiation of the requested I/O operation. A process in the *IO_init* state is represented by linking its PCB into an **IO_init queue**. If or when the device is ready to start the operation, an IOCB is set up for the requested operation, and the process is placed in the *IO_active* state. This is represented by linking its PCB into an **IO_active queue**. In most operating systems, the default strategy is to block processes waiting for I/O initiation or completion. Hence, once the I/O request is queued or started, the dispatcher is invoked to select the next process for execution. Typically the IO_init queue is organized as a linked list of PCBs, whereas the IO_active queue may be a linked list of IOCBs that point to PCBs. Much of I/O scheduling involves the manipulation of the linked list(s) that make up these two queues.

Figure 10-25 illustrates a structure using a single I/O queue for each state, and IOCBs to represent I/O activity on a disk and a printer. The I/O queues are designated `IO_initQ` and `IO_activeQ`, and the ready queue is also shown. In this example, PROCESS6, represented by PCB6, is in the *IO_active* state waiting for a disk I/O operation; PROCESS7, represented by PCB7, is in the *IO_active* state for a printer I/O operation. PROCESS1, PROCESS2, PROCESS3, PROCESS4, and PROCESS5, represented by PCB1, PCB2, PCB3, PCB4, and PCB5 respectively, are in the *IO_init* state waiting for initiation of I/O for the devices indicated.

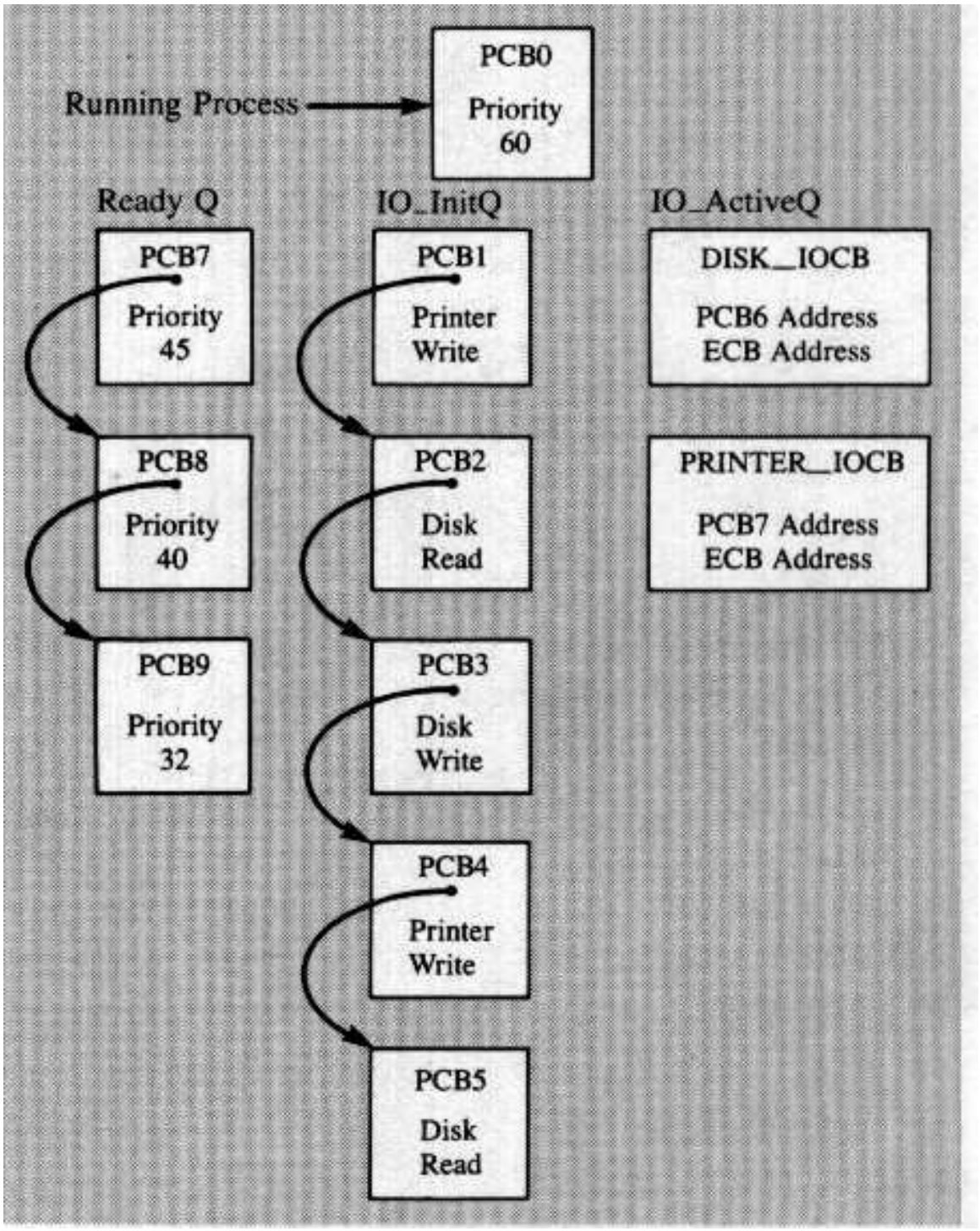


**Figure 10-25: Queue Structures for IO_init and IO_active States**

Suppose now that the disk operation requested by PROCESS6 completes. A disk I/O interrupt will be generated and the disk interrupt handler will receive control. This interrupt handler will determine that the operation is complete and invoke an algorithm similar to the one

shown in Figure 10-26. This algorithm results in PROCESS0, which was running, being returned to the *ready* state, PROCESS6 making a transition from the *IO_active* state to the *ready* state, and PROCESS2 making a transition from the *IO_init* state to the *IO_active* state.

```
IO_COMPLETE:

     /* Cleanup after disk operation,
          switch waiting process to ready state */
     if (operation is complete) {

          Save context of current process in PCB
          Insert PCB in ReadyQ
          Set Disk ECB for disk I/O just completed
          Move PCB address in Disk IOCB to ReadyQ
          Set Disk IOCB to idle state

     }

     /* setup next disk operation, if any */
     Search IO_WaitQ for another disk request
     if (disk request found) {

          Move PCB address of next disk I/O request to
               Disk IOCB
          Move DIsk ECB address to Disk IOCB
          Start requested disk I/O operation

     }

     Invoke dispatcher to dispatch next process
```

**Figure 10-26: Handling an I/O Request Complete Interrupt**

The result prior to the dispatching of the next process is shown in Figure 10-27. The transitions described are depicted in the expanded state diagram in Figure 10-28. We now have two blocked states, *IO_init* and *IO_active*, in place of the single state called *blocked*. Note that there is generally one IOCB for each I/O device (or perhaps controller) in the system. The IOCBs represent I/O in progress and identify the PCB of the process with I/O active.

*Device Scheduling Techniques*. Device scheduling is the process of selecting the next request to be processed by a device from one or more I/O queues. The request to be selected depends on the scheduling techniques being used. Most commonly, the selection is made by a simple first-in, first-out strategy. In other cases the priority of the requesting process might be used to determine which request to select. This method is necessary, for example, in real-time systems.

Special problems occur in scheduling the use of disk drives. Not only are there likely to be many requests pending in large systems, but the time required to service a request may depend heavily on the position on the disk to be accessed, especially if moving-head disks are used. The time required to perform a transfer with such disks is dominated by the seek time, which in turn depends on the distance between the track to be accessed and the track at which the disk access arm was previously positioned.

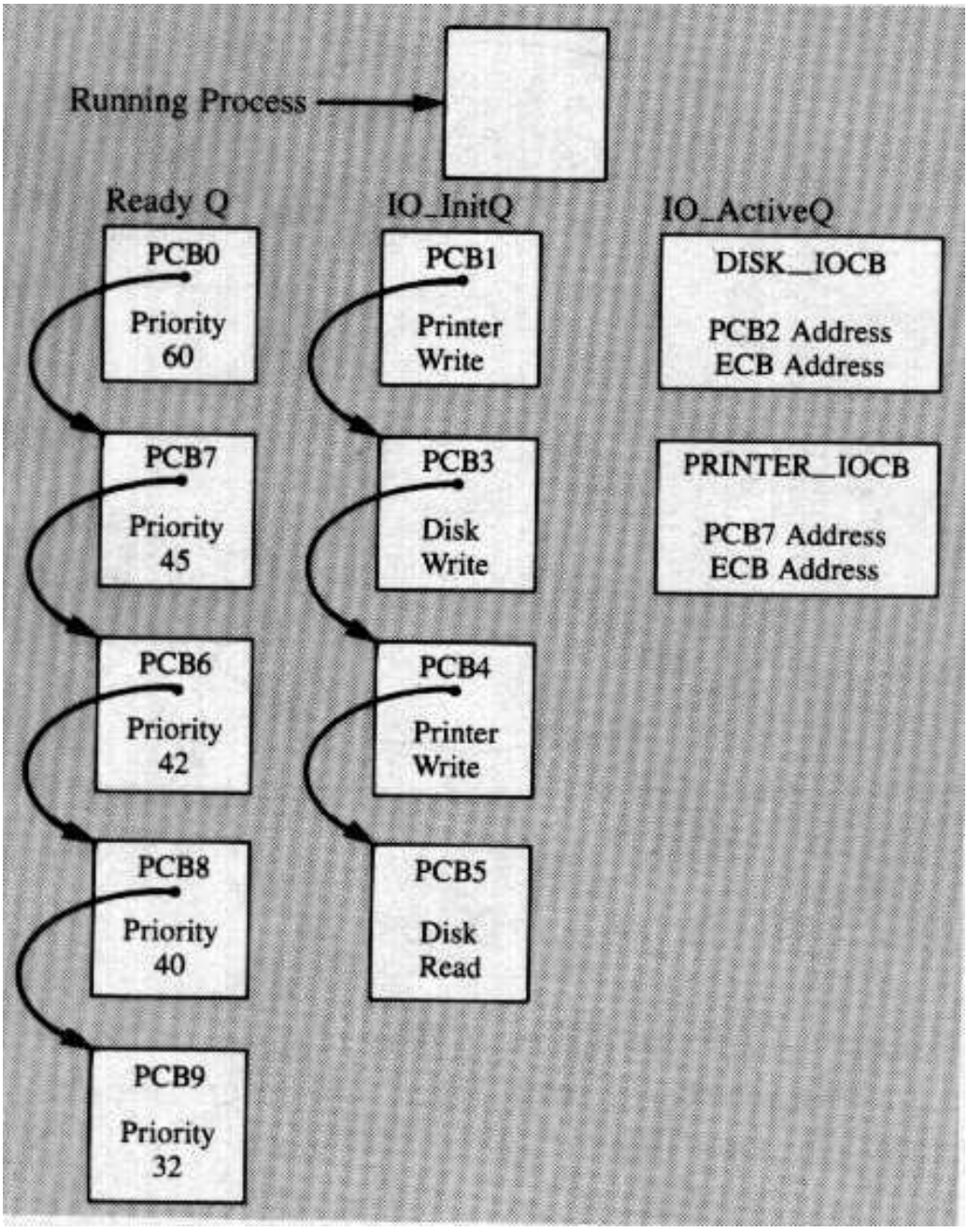


**Figure 10-27: Queues after I/O Completion for PROCESS6**

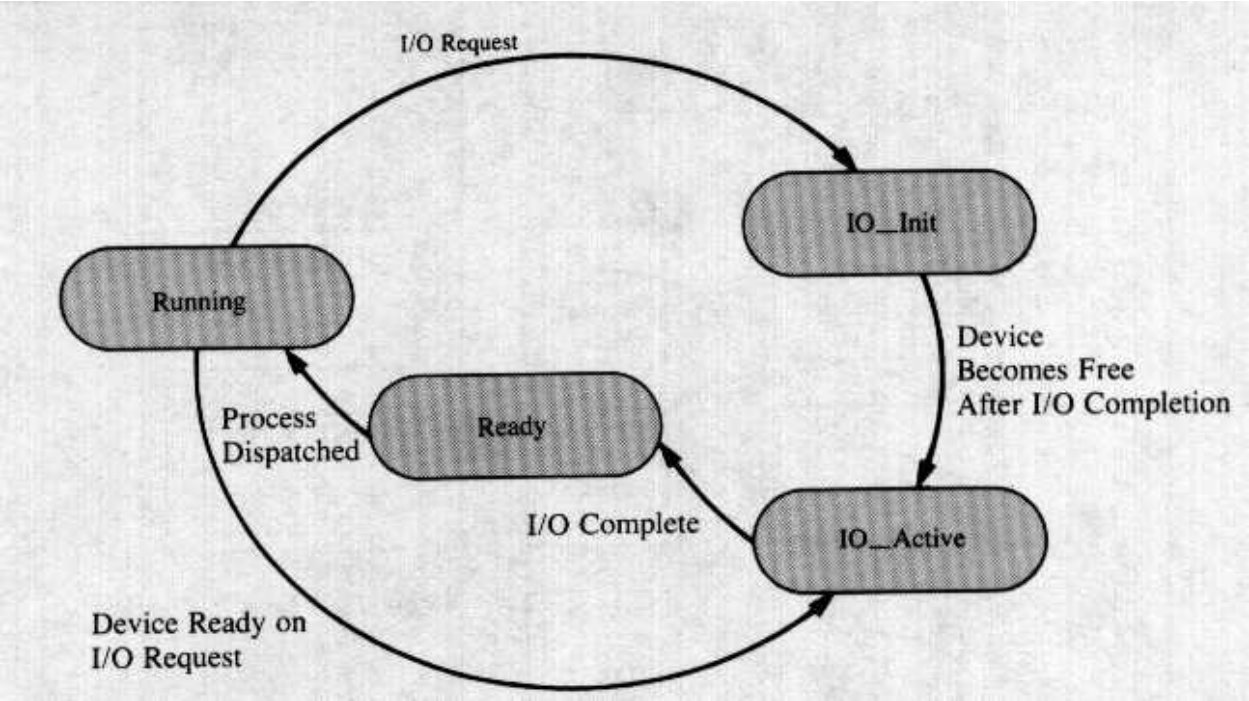


**Figure 10-28: State Transitions with IO_init and IO_active States**

Because of this situation, significant time may be saved by adopting a **shortest-seek-time-first** strategy, always selecting the request that causes the arm to move the shortest distance. However, problems arise similar to those encountered by shortest-job-next strategies for long-term process scheduling. Requests for access to locations close to the lowest or highest numbered tracks will be less likely to be close to the arm, and will tend to be discriminated against. In the extreme case, starvation is possible. What's more, the basis for discrimination, physical position of the data to be transferred, is not under control of processes and is not in any way a fair basis for assigning priorities.

Because of these problems, compromise methods are needed. The most common alternatives are based on requiring the arm to proceed in one direction until all pending requests are serviced before returning to deal with accesses in the other direction. These methods are further discussed, for example, by Peterson and Silberschatz [1985].

Some disks can sense rotational position, so that optimized scheduling could determine the request to schedule next by determining the sector that will come under the disk head soonest. This is an algorithm that would most likely be added to a fixed-head disk or drum.

I/O scheduling can become quite complex when optimization is added. The net benefit from such optimization is improved performance of the devices; however, it is often found in practice that the amount of improvement over a simple FIFO strategy is not great for the majority of systems.

## SUMMARY

Device management is a major responsibility of any operating system. This responsibility is most often met by a collection of device drivers. Usually a device driver is associated with each type of device. Because of the great variety of I/O devices, it is likely that new device drivers will need to be developed for individual installations. Thus developing device drivers is a likely activity for systems programmers.

Device drivers differ depending on the characteristics of individual devices. The structure and programming strategy for the I/O interface must also be considered. Key decisions include the choice of Simple I/O, block transfers using DMA, or I/O programs for I/O processors or channels. Interrupts must normally be used except in single-process environments, and other possible mechanisms include event flags or ECBs and various types of buffering.

Despite the variation, device drivers generally have a consistent structure and set of responsibilities. Most drivers consist of a main body and an interrupt handler. They are responsible for a series of standard activities, such as preparing for I/O, starting I/O, error handling, etc. They also must implement an I/O system call interface, providing support for a common set of I/O system calls.

I/O scheduling can become complex when many requests are pending from multiple processes. A system of queues is needed, and it may be useful to establish new process states depending on the status of pending I/O. Most requests are scheduled by a simple FIFO approach, but special problems arise in disk scheduling, where the time required to service a request may depend on the disk location to be accessed.

## FOR FURTHER READING

Device management has sometimes been neglected in OS texts. However, a reasonable treatment is contained in Tanenbaum [1987, Ch. 3]. A good older treatment of many important issues is presented in Chapters 2 and 5 of Madnick and Donovan [1974]. Extensive discussions are given by Finkel [1986. Ch. 5], and Milenkoviç[1987, Ch. 2].

For a thorough understanding of device drivers, there is no substitute for a study of actual implementations and OS guides. Device drivers in UNIX and VMS are fully described by Bach [1986] and Kenah and Bate [1984], respectively. The simple structure of the I/O subsystem in microprocessor OSs such as PC-DOS is reviewed by King [1983], among others. Some effective guides for writing device drivers are those produced by Digital Equipment for RT-11 [Digital 1981a], RSX-11 [Digital 1981b], and VMS [Digital 1984b]. Device drivers for MS-DOS are discussed by Wong [1986]. The art of writing Linux device drivers is presented by Rubini and Corbet [2001], available online at http://www.xml.com/ldd/chapter/book/bookindexpdf.html .

The topic of disk scheduling is thoroughly discussed in Chapter 7 of Peterson and Silberschatz [1985]. A more theoretical treatment is given in the classic text by Coffman and Denning [1973, Ch. 5].



-

## REVIEW QUESTIONS

1. List the items of information that must be supplied to a disk controller to set up a DMA read operation.

2. Give two reasons why it is especially important in smaller systems to have device drivers that are modular and easy to replace.

3. What are the disadvantages of disk (software) caching?

4. Why is polling of devices generally unacceptable in a multiprogramming operating system?

5. What functions are commonly provided by device drivers? Are there cases in which not all these functions are necessary? Why or why not?

6. What are the possible actions an operating system could take upon the detection of a disk error on a user disk read or write request? When and how would a process be informed of such a severe error?

## ASSIGNMENTS

1.. How would the use of a channel program change the structure and complexity of a device driver? What would be required of a printer process that sets up pages read from a spool file to be printed by a device driver, which will simply execute a channel program?

2. Modify the error recovery algorithm in Figure 10-21 to provide for the handling of the unexpected condition "backward at loadpoint" so that error recovery can continue.

3. Define data structures to represent I/O currently in progress on a device.

4. Compare device driver structures in two different operating systems (referring to the appropriate operating system manuals and textbooks). What implementation languages are used? How are interrupts used?

5. In a multiprogramming operating system, what happens when an I/O request is made by a process for a device that is busy with another I/O request?



-

# PROJECT

This section describes a three-part programming project to implement a set of device drivers for a simple multiprogramming operating system. You should read the general description in this text in conjunction with the more specific descriptions given in your project manual. These mechanisms are important components of the multiprogramming executive (MPX) project.

**a**. Design and implement a simple printer driver. The services supported by this device driver are:

1. `RC=PRT_OPEN(prt_flag):` Initialize the printer for operation. `prt_flag` is an event flag, and `RC` is a return code with a value of 0 if the requested operation is correct and has been initiated, or a negative value if an error has been detected. No I/O is initiated if `RC` is negative.

2. `RC=PRT_PRINT(buffer,length):` Print data on the printer. `buffer` is the data area from which characters are written; `length` specifies how many characters are to be printed. If the length parameter is omitted, the entire character string in buffer is printed.

3. `RC=PRT_CLOSE`: reset the printer, disabling interrupts.

**b**. Design and implement a simple device driver for a terminal connected via a serial interface. The driver must provide the following procedures, or services:

1. `RC=COM_OPEN(com_flag)`: Initialize the device for operation, enabling interrupts so that data can be captured in an internal "ring buffer" before it is needed by a process. Set `com_flag` as the event flag to be used for subsequent operations to determine if a requested operation has completed. `RC` is a return code , as described in part a.

2. `RC=COM_READ(buffer,length):` Read data from the device. `buffer` is the data area into which characters are read, and `length` specifies how many characters are to be read. Upon return, `RC` should indicate how many characters were read, if no error occurred; otherwise it should contain an error code as described above.

3. `RC=COM_WRITE(buffer,length):` Write data to the device. `buffer` is the data area from which characters are written; `length` specifies how many characters are to be written. If the length parameter is omitted, the entire character string in buffer is written to the device designated COM. Upon return, `RC` should indicate how many characters were written, if no error occurred; otherwise it should contain an error code.

4. `RC=COM_CLOSE`: Reset the device, disabling interrupts. Upon return, `RC` should have the value 0 if no error occurred; otherwise it should contain an error code.

**c**. Design and implement a low-level disk device driver. The services to be supported by this device driver are:

1. `RC=DISK_OPEN(disk_flag):` Initialize the disk for operation. `disk_flag` is an event flag, and `RC` is a return code, as described in part a.

2. `RC=DISK_READ(track,sector,buffer,length):` Read a block of data from the disk, beginning at the specified track and sector. The location to copy the data to is given by `buffer` and the size of the block is specified by `length`. This must be less than or equal to the sector size. The return code is used as described in Part a.

3. `RC=DISK_WRITE(track,sector,buffer,length):` write a block of data to the disk, beginning at the specified track and sector. The location to copy the data from is given by `buffer` and the size of the block is specified by `length`. This must be less than or equal to the sector size. The return code is used as described in Part a.

4. `RC=DISK_CLOSE`: reset the disk to an idle state.

5. `RC=DISK_STATUS(status):` set the value of `status` to the current hardware status of the disk.