One of the main functions of an operating system is to control all the computer's I/O (Input/Output) devices. It must issue commands to the devices, catch interrupts, and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. To the extent possible, the interface should be the same for all devices (device independence). The I/O code represents a significant fraction of the total operating system. How the operating system manages I/O is the subject of this chapter.

An outline of the chapter is as follows. First we will look briefly at some of the principles of I/O hardware, and then we will look at I/O software in general. I/O software can be structured in layers, with each layer having a well-defined task to perform. We will look at these layers to see what they do and how they fit together.

After that comes a section on deadlocks. We will define deadlocks precisely, show how they are caused, give two models for analyzing them, and discuss some algorithms for preventing their occurrence.

Then we will take a bird's-eye view of I/O in MINIX. Following that introduction, we will look at four I/O devices in detail—the RAM disk, the hard disk, the clock, and the terminal. For each device we will look at its hardware, software, and implementation in MINIX. Finally, the chapter closes with a short discussion of a little piece of MINIX that is located in the same layer as the I/O tasks but is itself not an I/O task. It provides some services to the memory manager and file system, such as fetching blocks of data from a user process.
3.1 PRINCIPLES OF I/O HARDWARE

Different people look at I/O hardware in different ways. Electrical engineers look at it in terms of chips, wires, power supplies, motors, and all the other physical components that make up the hardware. Programmers look at the interface presented to the software—the commands the hardware accepts, the functions it carries out, and the errors that can be reported back. In this book we are concerned with programming I/O devices, not designing, building, or maintaining them, so our interest will be restricted to how the hardware is programmed, not how it works inside. Nevertheless, the programming of many I/O devices is often intimately connected with their internal operation. In the next three sections we will provide a little general background on I/O hardware as it relates to programming.

3.1.1 I/O Devices

I/O devices can be roughly divided into two categories: block devices and character devices. A block device is one that stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 bytes to 32,768 bytes. The essential property of a block device is that it is possible to read or write each block independently of all the other ones. Disks are the most common block devices.

If you look closely, the boundary between devices that are block addressable and those that are not is not well defined. Everyone agrees that a disk is a block addressable device because no matter where the arm currently is, it is always possible to seek to another cylinder and then wait for the required block to rotate under the head. Now consider an 8mm or DAT tape drive used for making disk backups. Its tapes generally contain fixed-size blocks. If the tape drive is given a command to read block N, it can always rewind the tape and go forward until it comes to block N. This operation is analogous to a disk doing a seek, except that it takes much longer. Also, it may or may not be possible to rewrite one block in the middle of a tape. Even if it were possible to use tapes as random access block devices, that is stretching the point somewhat: they are normally not used that way.

The other type of I/O device is the character device. A character device delivers or accepts a stream of characters, without regard to any block structure. It is not addressable and does not have any seek operation. Printers, network interfaces, mice (for pointing), rats (for psychology lab experiments), and most other devices that are not disk-like can be seen as character devices.

This classification scheme is not perfect. Some devices just do not fit in. Clocks, for example, are not block addressable. Nor do they generate or accept character streams. All they do is cause interrupts at well-defined intervals. Memory-mapped screens do not fit the model well either. Still, the model of
block and character devices is general enough that it can be used as a basis for making some of the operating system software dealing with I/O device independent. The file system, for example, deals just with abstract block devices and leaves the device-dependent part to lower-level software called device drivers.

### 3.1.2 Device Controllers

I/O units typically consist of a mechanical component and an electronic component. It is often possible to separate the two portions to provide a more modular and general design. The electronic component is called the device controller or adapter. On personal computers, it often takes the form of a printed circuit card that can be inserted into a slot on the computer's parentboard (previously incorrectly called a motherboard). The mechanical component is the device itself.

The controller card usually has a connector on it, into which a cable leading to the device itself can be plugged. Many controllers can handle two, four, or even eight identical devices. If the interface between the controller and device is a standard interface, either an official standard such as ANSI, IEEE, or ISO, or a de facto one, then companies can make controllers or devices that fit that interface. Many companies, for example, make disk drives that match the IDE (Integrated Drive Electronics) or SCSI (Small Computer System Interface) disk controller interfaces.

We mention this distinction between controller and device because the operating system nearly always deals with the controller, not the device. Most small computers use the single bus model of Fig. 3-1 for communication between the CPU and the controllers. Large mainframes often use a different model, with multiple buses and specialized I/O computers called I/O channels taking some of the load off the main CPU.

![Figure 3-1. A model for connecting the CPU, memory, controllers, and I/O devices.](image)

The interface between the controller and the device is often a very low-level interface. A disk, for example, might be formatted with 16 sectors of 512 bytes per track. What actually comes off the drive, however, is a serial bit stream, starting with a preamble, then the 4096 bits in a sector, and finally a checksum, also called an Error-Correcting Code (ECC). The preamble is written when the disk
is formatted and contains the cylinder and sector number, the sector size, and the block address of the data, as well as synchronization information.

The controller's job is to convert the serial bit stream into a block of bytes and perform any error correction necessary. The block of bytes is typically first assembled, bit by bit, in a buffer inside the controller. After the checksum has been verified and the block declared to be error free, it can then be transmitted.
commands have parameters, which are also loaded into the controller's registers. When a command has been accepted, the CPU can leave the controller alone and go off to do other work. When the command has been completed, the controller causes an interrupt in order to allow the operating system to gain control of the CPU and test the results of the operation. The CPU gets the results and device status by reading one or more bytes of information from the controller's registers.

3.1.3 Direct Memory Access (DMA)

Many controllers, especially those for block devices, support Direct Memory Access or DMA. To explain how DMA works, let us first look at how disk reads occur when DMA is not used. First the controller reads the block (one or more sectors) from the drive serially, bit by bit, until the entire block is in the controller's internal buffer. Next, it computes the checksum to verify that no read errors have occurred. Then the controller causes an interrupt. When the operating system starts running, it can read the disk block from the controller's buffer a byte or a word at a time by executing a loop, with each iteration reading one byte or word from a controller device register and storing it in memory.

Naturally, a programmed CPU loop to read the bytes one at a time from the controller wastes CPU time. DMA was invented to free the CPU from this low-level work. When it is used, the CPU gives the controller two items of information, in addition to the disk address of the block: the memory address where the block is to go, and the number of bytes to transfer, as shown in Fig. 3-3.

After the controller has read the entire block from the device into its buffer and verified the checksum, it copies the first byte or word into the main memory at the address specified by the DMA memory address. Then it increments the DMA address and decrements the DMA count by the number of bytes just transferred. This process is repeated until the DMA count becomes zero, at which time
Figure 3.3. A DMA transfer is done entirely by the controller.

the controller causes an interrupt. When the operating system starts up, it does not have to copy the block to memory; it is already there.

You may be wondering why the controller does not just store the bytes in main memory as soon as it gets them from the disk. In other words, why does it need an internal buffer? The reason is that once a disk transfer has started, the bits keep arriving from the disk at a constant rate, whether the controller is ready for them or not. If the controller tried to write data directly to memory, it would have to go over the system bus for each word transferred. If the bus were busy due to some other device using it, the controller would have to wait. If the next disk word arrived before the previous one had been stored, the controller would have to store it somewhere. If the bus were very busy, the controller might end up storing quite a few words and having a lot of administration to do as well. When the block is buffered internally, the bus is not needed until the DMA begins, so the design of the controller is much simpler because the DMA transfer to memory is not time critical. (Some older controllers did, in fact, go directly to memory with only a small amount of internal buffering, but when the bus was very busy, a transfer might have had to be terminated with an overrun error.)

The two-step buffering process described above has important implications for I/O performance. While the data are being transferred from the controller to the memory, either by the CPU or by the controller, the next sector will be passing under the disk head and the bits arriving in the controller. Simple controllers just cannot cope with doing input and output at the same time, so while a memory transfer is taking place, the sector passing under the disk head is lost.

As a result, the controller will be able to read only every other block. Reading a complete track will then require two full rotations, one for the even blocks and one for the odd blocks. If the time to transfer a block from the controller to memory over the bus is longer than the time to read a block from the disk, it may be necessary to read one block and then skip two (or more) blocks.

Skipping blocks to give the controller time to transfer data to memory is called interlaving. When the disk is formatted, the blocks are numbered to take account of this, and when a block is read, the next one is read, and so on, leaving.

3.2 PROMOTION

The first problem with the read controller is that it requires a lot of bits which the controller has to manage. The controller usually has an 8-bit data bus, but reading an 8-bit word from the disk is more than the problem. The controller must be able to handle the rest of the hardware, but it does not want to handle this too.

Not only is this not practical, but it also often falls short of what is required. The limiting factor is the bus: 8 bits is the maximum it can do, hence the slower the disk is, the slower it is. Also, there is often no way to fix this in software.

3.2.1 Growth

A key problem with promotion is the size of the file. As files grow, the number of sectors the file contains increases, and this increases the number of times the controller has to read a particular sector. This can lead to a situation where the controller is not able to keep up with the read requests from the disk. This is known as the promotion problem. To avoid this problem, the controller must be able to keep up with the read requests from the disk. This is accomplished by buffering the read requests from the disk and then reading the sector when the controller has the bandwidth to do so.
account of the interleave factor. In Fig. 3-4(a) we see a disk with 8 blocks per track and no interleaving. In Fig. 3-4(b) we see the same disk with single interleaving. In Fig. 3-4(c) double interleaving is shown.

Figure 3-4. (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

The idea of numbering the blocks this way is to allow the operating system to read consecutively numbered blocks and still achieve the maximum speed of which the hardware is capable. If the blocks were numbered as in Fig. 3-4(a) but the controller could read only alternate blocks, an operating system that allocated an 8-block file in consecutive disk blocks would require eight disk rotations to read blocks 0 through 7 in order. (Of course, if the operating system knew about the problem and allocated its blocks differently, it could solve the problem in software, but it is better to have the controller worry about the interleaving.)

Not all computers use DMA. The argument against it is that the main CPU is often far faster than the DMA controller and can do the job much faster (when the limiting factor is not the speed of the I/O device). If there is no other work for it to do, having the (fast) CPU wait for the (slow) DMA controller to finish is pointless. Also, getting rid of the DMA controller and having the CPU do all the work in software saves some money.

3.2 PRINCIPLES OF I/O SOFTWARE

Let us turn away from the hardware and now look at how the I/O software is structured. The general goals of the I/O software are easy to state. The basic idea is to organize the software as a series of layers, with the lower ones concerned with hiding the peculiarities of the hardware from the upper ones, and the upper ones concerned with presenting a nice, clean, regular interface to the users. In the following sections we will look at these goals and how they are achieved.

3.2.1 Goals of the I/O Software

A key concept in the design of I/O software is known as device independence. What it means is that it should be possible to write programs that can read files on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify
the programs for each different device type. One should be able to type a command such as

```
sort <input >output
```

and have it work with input coming from a floppy disk, a hard disk, or the keyboard, and the output going to the floppy disk, the hard disk, or even the screen. It is up to the operating system to take care of the problems caused by the fact that these devices really are different and require very different device drivers to actually write the data to the output device.

Closely related to device independence is the goal of **uniform naming.** The name of a file or a device should simply be a string or an integer and not depend on the device in any way. In UNIX, all disks can be integrated together in the file system hierarchy in arbitrary ways so the user need not be aware of which name corresponds to which device. For example, a floppy disk can be mounted on top of the directory `/usr/lst/backup` so that copying a file to `/usr/lst/backup/monday` copies the file to the floppy disk. In this way, all files and devices are addressed the same way: by a path name.

Another important issue for I/O software is error handling. In general, errors should be handled as close to the hardware as possible. If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it, perhaps by just trying to read the block again. Many errors are transient, such as read errors caused by specks of dust on the read head, and will go away if the operation is repeated. Only if the lower layers are not able to deal with the problem should the upper layers be told about it. In many cases, error recovery can be done transparently at a low level without the upper levels even knowing about the error.

Still another key issue is synchronous (blocking) versus asynchronous (interrupt-driven) transfers. Most physical I/O is asynchronous—the CPU starts the transfer and goes off to do something else until the interrupt arrives. User programs are much easier to write if the I/O operations are blocking—after a READ command the program is automatically suspended until the data are available in the buffer. It is up to the operating system to make operations that are actually interrupt-driven look blocking to the user programs.

The final concept that we will deal with here is sharable versus dedicated devices. Some I/O devices, such as disks, can be used by many users at the same time. No problems are caused by multiple users having open files on the same disk at the same time. Other devices, such as tape drives, have to be dedicated to a single user until that user is finished. Then another user can have the tape drive. Having two or more users writing blocks intermixed at random to the same tape will definitely not work. Introducing dedicated (unshared) devices also introduces a variety of problems. Again, the operating system must be able to handle both shared and dedicated devices in a way that avoids problems.

These four points summarize the I/O system.

### 3.2.2 Interrupts

Interrupts form the backbone of UNIX. Each process knows about disk I/O operations. When an I/O operation begins, the process can have control transferred to a special variable, or it can be blocked.

When an I/O operation is done, the kernel interrupts the process. semaphores, which are used to pass information, and still others which are used to pass control. The effect of these semaphores is that the process can be able to

### 3.2.3 Devices

All the devices in UNIX are handled by **device drivers.** For example, it would be easy to write a device driver for a remote device and call the system services to do the I/O. On the other hand, it would be much more complex to write a map graph of a remote device and call the device driver to be used.

Earlier in this chapter, we used the device driver to represent each control block. The device drivers for dedicated devices and device drivers for dedicated devices are not used in the same way. Thus, the

SEC. 3.2

These four points summarize the I/O system.

1. **Uniform naming.**
2. **Error handling.**
3. **Synchronous versus asynchronous transfers.**
4. **Shared versus dedicated devices.**

These four points summarize the I/O system.

### 3.2.2 Interrupts

Interrupts form the backbone of UNIX. Each process knows about disk I/O operations. When an I/O operation begins, the process can have control transferred to a special variable, or it can be blocked.

When an I/O operation is done, the kernel interrupts the process. semaphores, which are used to pass information, and still others which are used to pass control. The effect of these semaphores is that the process can be able to

### 3.2.3 Devices

All the devices in UNIX are handled by **device drivers.** For example, it would be easy to write a device driver for a remote device and call the system services to do the I/O. On the other hand, it would be much more complex to write a map graph of a remote device and call the device driver to be used.

Earlier in this chapter, we used the device driver to represent each control block. The device drivers for dedicated devices and device drivers for dedicated devices are not used in the same way. Thus, the
These goals can be achieved in a comprehensible and efficient way by structuring the I/O software in four layers:

1. Interrupt handlers (bottom).
2. Device drivers.
3. Device-independent operating system software.
4. User-level software (top).

These four layers are (not accidentally) the same four layers that we saw in Fig. 2-26. In the following sections we will look at each one in turn, starting at the bottom. The emphasis in this chapter is on the device drivers (layer 2), but we will summarize the rest of the I/O software to show how the various pieces of the I/O system fit together.

### 3.2.2 Interrupt Handlers

Interrupts are an unpleasant fact of life. They should be hidden away, deep in the bowels of the operating system, so that as little of the system as possible knows about them. The best way to hide them is to have every process starting an I/O operation block until the I/O has completed and the interrupt occurs. The process can block itself by doing a `DOWN` on a semaphore, a `WAIT` on a condition variable, or a `RECEIVE` on a message, for example.

When the interrupt happens, the interrupt procedure does whatever it has to in order to unblock the process that started it. In some systems it will do an `UP` on a semaphore. In others it will do a `SIGNAL` on a condition variable in a monitor. In still others, it will send a message to the blocked process. In all cases the net effect of the interrupt will be that a process that was previously blocked will now be able to run.

### 3.2.3 Device Drivers

All the device-dependent code goes in the device drivers. Each device driver handles one device type, or at most, one class of closely related devices. For example, it would probably be a good idea to have a single terminal driver, even if the system supported several different brands of terminals, all slightly different. On the other hand, a dumb, mechanical hardcopy terminal and an intelligent bitmap graphics terminal with a mouse are so different that different drivers should be used.

Earlier in this chapter we looked at what device controllers do. We saw that each controller has one or more device registers used to give it commands. The device drivers issue these commands and check that they are carried out properly. Thus, the disk driver is the only part of the operating system that knows how
many registers that disk controller has and what they are used for. It alone knows about sectors, tracks, cylinders, heads, arm motion, interleave factors, motor drives, head settling times, and all the other mechanics of making the disk work properly.

In general terms, the job of a device driver is to accept abstract requests from the device-independent software above it and see to it that the request is executed. A typical request is to read block \( n \). If the driver is idle at the time a request comes in, it starts carrying out the request immediately. If, however, it is already busy with a request, it will normally enter the new request into a queue of pending requests to be dealt with as soon as possible.

The first step in actually carrying out an I/O request, say, for a disk, is to translate it from abstract to concrete terms. For a disk driver, this means figuring out where on the disk the requested block actually is, checking to see if the drive's motor is running, determining if the arm is positioned on the proper cylinder, and so on. In short, it must decide which controller operations are required and in what sequence.

Once it has determined which commands to issue to the controller, it starts issuing them by writing into the controller's device registers. Some controllers can handle only one command at a time. Other controllers are willing to accept a linked list of commands, which they then carry out by themselves without further help from the operating system.

After the command or commands have been issued, one of two situations will apply. In many cases the device driver must wait until the controller does some work for it, so it blocks itself until the interrupt comes in to unblock it. In other cases, however, the operation finishes without delay, so the driver need not block. As an example of the latter situation, scrolling the screen on some terminals requires just writing a few bytes into the controller's registers. No mechanical motion is needed, so the entire operation can be completed in a few microseconds.

In the former case, the blocked driver will be awakened by the interrupt. In the latter case, it will never go to sleep. Either way, after the operation has been completed, it must check for errors. If everything is all right, the driver may have data to pass to the device-independent software (e.g., a block just read). Finally, it returns some status information for error reporting back to its caller. If any other requests are queued, one of them can now be selected and started. If nothing is queued, the driver blocks waiting for the next request.

### 3.2.4 Device-Independent I/O Software

Although some of the I/O software is device specific, a large fraction of it is device independent. The exact boundary between the drivers and the device-independent software is system dependent, because some functions that could be done in a device-independent way may actually be done in the drivers, for efficiency or other reasons. The functions shown in Fig. 3-5 are typically done in the

--

The basic file system functions that are part of the user-level layer are:

- `mkdir`: Create a new directory.
- `chmod`: Change file permissions.
- `chown`: Change file ownership.

A major device class is network devices. For example, the device node `/dev/tty00` corresponds to the major device node `/dev/tty0`. Each of these device nodes also corresponds to a network interface or a serial port in the driver that serves it.

Closely tied to the file system is network access. In most users' minds, a computer system is used to access the network. In many cases, it is the network that is accessed, and there is no connection to the user-level layer.

Different independent software systems, for example, use different operating system services. This is why modern systems have a larger unit.
device-independent software. In Minix, most of the device-independent software is part of the file system, in layer 3 (Fig. 2.26). Although we will study the file system in Chap. 5, we will take a quick look at the device-independent software here, to provide some perspective on I/O and show better where the drivers fit in.

| Uniform interfacing for device drivers |
| Device naming                          |
| Device protection                      |
| Providing a device-independent block size |
| Buffering                               |
| Storage allocation on block devices     |
| Allocating and releasing dedicated devices |
| Error reporting                        |

Figure 3.5. Functions of the device-independent I/O software.

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software.

A major issue in an operating system is how objects such as files and I/O devices are named. The device-independent software takes care of mapping symbolic device names onto the proper driver. In Unix, a device name, such as /dev/tty00, uniquely specifies the i-node for a special file, and this i-node contains the major device number, which is used to locate the appropriate driver. The i-node also contains the minor device number, which is passed as a parameter to the driver to specify the unit to be read or written.

Closely related to naming is protection. How does the system prevent users from accessing devices that they are not entitled to access? In most personal computer systems, there is no protection at all. Any process can do anything it wants to. In most mainframe systems, access to I/O devices by user processes is completely forbidden. In Unix, a more flexible scheme is used. The special files corresponding to I/O devices are protected by the usual rwx bits. The system administrator can then set the proper permissions for each device.

Different disks may have different sector sizes. It is up to the device-independent software to hide this fact and provide a uniform block size to higher layers, for example, by treating several sectors as a single logical block. In this way, the higher layers only deal with abstract devices that all use the same logical block size, independent of the physical sector size. Similarly, some character devices deliver their data one byte at a time (e.g., modems), while others deliver theirs in larger units (e.g., network interfaces). These differences must also be hidden.
Buffering is also an issue, both for block and character devices. For block devices, the hardware generally insists upon reading and writing entire blocks at once, but user processes are free to read and write in arbitrary units. If a user process writes half a block, the operating system will normally keep the data around internally until the rest of the data are written, at which time the block can go out to the disk. For character devices, users can write data to the system faster than it can be output, necessitating buffering. Keyboard input that arrives before it is needed also requires buffering.

When a file is created and filled with data, new disk blocks have to be allocated to the file. To perform this allocation, the operating system needs a list or bit map of free blocks per disk, but the algorithm for locating a free block is device independent and can be done above the level of the driver.

Some devices, such as CD-ROM recorders, can be used only by a single process at any given moment. It is up to the operating system to examine requests for device usage and accept or reject them, depending on whether the requested device is available or not. A simple way to handle these requests is to require processes to perform Opens on the special files for devices directly. If the device is unavailable, the OPEN will fail. Closing such a dedicated device would then release it.

Error handling, by and large, is done by the drivers. Most errors are highly device dependent, so only the driver knows what to do (e.g., retry, ignore it, panic). A typical error is caused by a disk block that has been damaged and cannot be read any more. After the driver has tried to read the block a certain number of times, it gives up and informs the device-independent software. How the error is treated from here on is device independent. If the error occurred while reading a user file, it may be sufficient to report the error back to the caller. However, if it occurred while reading a critical system data structure, such as the block containing the bit map showing which blocks are free, the operating system may have no choice but to print an error message and terminate.

### 3.2.5 User-Space I/O Software

Although most of the I/O software is within the operating system, a small portion of it consists of libraries linked together with user programs, and even whole programs running outside the kernel. System calls, including the I/O system calls, are normally made by library procedures. When a C program contains the call

```c
count = write(fd, buffer, nbytes);
```

the library procedure `write` will be linked with the program and contained in the binary program present in memory at run time. The collection of all these library procedures is clearly part of the I/O system.

While these procedures do little more than put their parameters in the appropriate place for the system call, there are other I/O procedures that actually do real work. The `write` procedure and some variations on it store a string. Another procedure stores it in a `buffer`.

The standard procedure used to run as part of the system

Not all of the important categories of I/O devices are files, such as the printer. A character special file, such as terminal 0, nothing for the device.

Instead of having a special directory that contains the device as a special file, two character special file, terminal 0, direct use by the system, and all is eliminated.

Spooling, for example, files from a printer somewhere into the daemon takes a long time is the system. As the machines are zoned, and mail cannot be sent mail to Bob, the file is spooled and the entire mail system

Figure 3.2 illustrates the normal function of the I/O routines and what happens when a user process...

The arrow shows how the `write` routine read a block and wrote to the disk. The other arrow shows that if the needed block is not found in the hardware, the I/O routine is not completed.

When the I/O device is being handled, the kernel is running under the direction of the current process.
do real work. In particular, formatting of input and output is done by library procedures. One example from C is `printf`, which takes a format string and possibly some variables as input, builds an ASCII string, and then calls `write` to output the string. An example of a similar procedure for input is `scanf` which reads input and stores it into variables described in a format string using the same syntax as `printf`.

The standard I/O library contains a number of procedures that involve I/O and all run as part of user programs.

Not all user-level I/O software consists of library procedures. Another important category is the spooling system. **Spooling** is a way of dealing with dedicated I/O devices in a multiprogramming system. Consider a typical spooled device: a printer. Although it would be technically easy to let any user process open the character special file for the printer, suppose a process opened it and then did nothing for hours. No other process could print anything.

Instead what is done is to create a special process, called a **daemon**, and a special directory, called a **spooling directory**. To print a file, a process first generates the entire file to be printed and puts it in the spooling directory. It is up to the daemon, which is the only process having permission to use the printer’s special file, to print the files in the directory. By protecting the special file against direct use by users, the problem of having someone keeping it open unnecessarily long is eliminated.

Spooling is not only used for printers. It is also used in other situations. For example, file transfer over a network often uses a network daemon. To send a file somewhere, a user puts it in a network spooling directory. Later on, the network daemon takes it out and transmits it. One particular use of spooled file transmission is the Internet electronic mail system. This network consists of millions of machines around the world communicating using many computer networks. To send mail to someone, you call a program such as `send`, which accepts the letter to be sent and then deposits it in a spooling directory for transmission later. The entire mail system runs outside the operating system.

Figure 3-6 summarizes the I/O system, showing all the layers and the principal functions of each layer. Starting at the bottom, the layers are the hardware, interrupt handlers, device drivers, device-independent software, and finally the user processes.

The arrows in Fig. 3-6 show the flow of control. When a user program tries to read a block from a file, for example, the operating system is invoked to carry out the call. The device-independent software looks in the block cache, for example. If the needed block is not there, it calls the device driver to issue the request to the hardware. The process is then blocked until the disk operation has been completed.

When the disk is finished, the hardware generates an interrupt. The interrupt handler is run to discover what has happened, that is, which device wants attention right now. It then extracts the status from the device and wakes up the sleeping process to finish off the I/O request and let the user process continue.
3.3 DEADLOCKS

Computer systems are full of resources that can only be used by one process at a time. Common examples include flatbed plotters, CD-ROM readers, CD-ROM recorders, 8mm DAT tape drive backup systems, imagesetters, and slots in the system’s process table. Having two processes simultaneously writing to the printer leads to gibberish. Having two processes using the same slot in the process table will probably lead to a system crash. Consequently, all operating systems have the ability to (temporarily) grant a process exclusive access to certain resources.

For many applications, a process needs exclusive access to not one resource, but several. Consider, for example, a marketing company that specializes in making large, detailed demographic maps of the United States on a 1-meter wide flatbed plotter. The demographic information comes from CD-ROMs containing census and other data. Suppose that process A asks for the CD-ROM drive and gets it. A moment later, process B asks for the flatbed plotter and gets it, too. Now process A asks for the plotter and blocks waiting for it. Finally, process B asks for the CD-ROM drive and also blocks. At this point both processes are blocked and will remain so forever. This situation is called a deadlock.

Deadlocks are not a good thing to have in your system.

Deadlocks can occur in many situations besides requesting dedicated I/O devices. In a database system, for example, a program may have to lock several records it is using, to avoid race conditions. If process A locks record R1 and process B locks record R2, and then each process tries to lock the other one’s record, we also have a deadlock. Thus deadlocks can occur on hardware resources or on software resources.

In this section we will examine deadlocks more closely to see how they arise and how they can be prevented or avoided. As examples, we will talk about acquiring resources that are not available, because this is what often happens. Were not concerned with the problem of acquiring resources that are available, any of them, because this is a resource inordinately

3.3.1 Resources

Deadlocks occur because processes request access to resources. If any possible, it can lead to a situation where the system cannot proceed, because it starts to consume more resources than it needs. On the other hand, if the system is available, it cannot be used, any of them, because this is a resource inordinately.

Resources

Resources are used to accomplish tasks. In general, they are available in limited quantities. For example, a system may have only one disk drive, so that each process that requests access to the drive starts to consume more resources than it needs.

Processes

The processes that use the resources are started by the system. Initially, we need only one process: the main memory, a process that, unfortunately, is only in use and switched to another process.

A non-

A non-process is a process that is not in use. A process that is not in use begins to process one resource at a time. The process will begin to process the next resource when the current resource is finished.

In general, a process that would ordinarily involve more than one resource from one process to another, the resources.

The section 3.3.1 resources discusses

1. Resources
2. Usage
3. Resources

If the resource

The resources

The resources

The resources

The resources

The resources

The resources

The resources

The resources

The resources