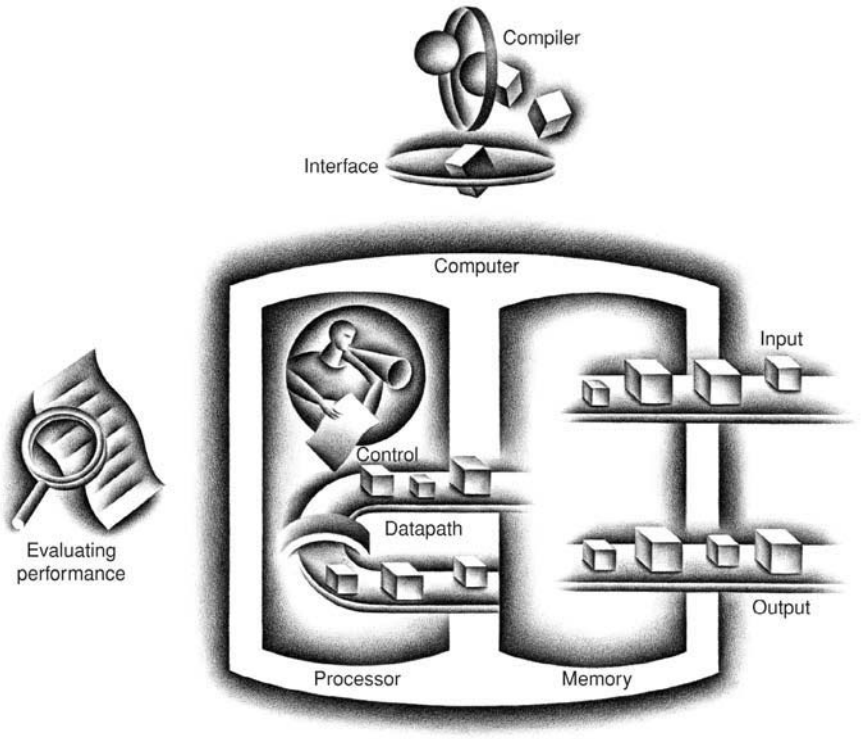
8

Storage, Networks, and Other Peripherals

Combining bandwidth and storage . . . enables swift and reliable access to the ever-expanding troves of content on the proliferating disks and . . . repositories of the Internet.

George Gilder
The End Is Drawing Nigh, 2000

The Five Classic Components of a Computer



8.1 Introduction

Although users can get frustrated if their computer hangs and must be rebooted, they become apoplectic if their storage system crashes and they lose information. Thus, the bar for dependability is much higher for storage than for computation. Networks also plan for failures in communication, including several mechanisms to detect and recover from such failures. Hence, I/O systems generally place much greater emphasis on dependability and cost, while processors and memory focus on performance and cost.

I/O systems must also plan for expandability and for diversity of devices, which is not a concern for processors. Expandability is related to storage capacity, which is another design parameter for I/O systems; systems may need a lower bound of storage capacity to fulfill their role.

Although performance plays a smaller role for I/O, it is more complex. For example, with some devices we may care primarily about access latency, while with others throughput is crucial. Furthermore, performance depends on many aspects of the system: the device characteristics, the connection between the device and the rest of the system, the memory hierarchy, and the operating system. Figure 8.1 shows the structure of a simple system with its I/O. All of the components, from the individual I/O devices to the processor to the system software, will affect the dependability, expandability, and performance of tasks that include I/O.

I/O devices are incredibly diverse. Three characteristics are useful in organizing this wide variety:

- *Behavior:* Input (read once), output (write only, cannot be read), or storage (can be reread and usually rewritten).
- *Partner:* Either a human or a machine is at the other end of the I/O device, either feeding data on input or reading data on output.
- *Data rate:* The peak rate at which data can be transferred between the I/O device and the main memory or processor. It is useful to know what maximum demand the device may generate.

For example, a keyboard is an *input* device used by a *human* with a *peak data rate* of about 10 bytes per second. Figure 8.2 shows some of the I/O devices connected to computers.

In Chapter 1, we briefly discussed four important and characteristic I/O devices: mice, graphics displays, disks, and networks. In this chapter we go into much more depth on disk storage and networks.

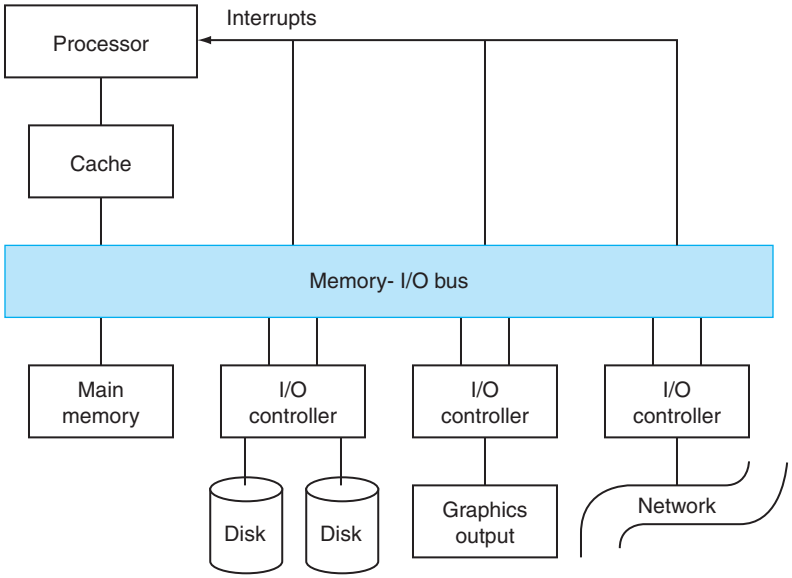


FIGURE 8.1 A typical collection of I/O devices. The connections between the I/O devices, processor, and memory are usually called *buses*. Communication among the devices and the processor use both interrupts and protocols on the bus, as we will see in this chapter. Figure 8.11 on page 585 shows the organization for a desktop PC.

How we should assess I/O performance often depends on the application. In some environments, we may care primarily about system throughput. In these cases, I/O bandwidth will be most important. Even I/O bandwidth can be measured in two different ways:

1. How much data can we move through the system in a certain time?
2. How many I/O operations can we do per unit of time?

Which performance measurement is best may depend on the environment. For example, in many multimedia applications, most I/O requests are for long streams of data, and transfer bandwidth is the important characteristic. In another environment, we may wish to process a large number of small, unrelated accesses to an I/O device. An example of such an environment might be a tax-processing office of the National Income Tax Service (NITS). NITS mostly cares about processing a large number of forms in a given time; each tax form is stored separately and is fairly small. A system oriented toward large file transfer may be satisfactory, but an I/O system that can support the simultaneous transfer of many small files may be cheaper and faster for processing millions of tax forms.

Device	Behavior	Partner	Data rate (Mbit/sec)
Keyboard	input	human	0.0001
Mouse	input	human	0.0038
Voice input	input	human	0.2640
Sound input	input	machine	3.0000
Scanner	input	human	3.2000
Voice output	output	human	0.2640
Sound output	output	human	8.0000
Laser printer	output	human	3.2000
Graphics display	output	human	800.0000–8000.0000
Modem	input or output	machine	0.0160–0.0640
Network/LAN	input or output	machine	100.0000–1000.0000
Network/wireless LAN	input or output	machine	11.0000–54.0000
Optical disk	storage	machine	80.0000
Magnetic tape	storage	machine	32.0000
Magnetic disk	storage	machine	240.0000–2560.0000

FIGURE 8.2 The diversity of I/O devices. I/O devices can be distinguished by whether they serve as input, output, or storage devices; their communication partner (people or other computers); and their peak communication rates. The data rates span eight orders of magnitude. Note that a network can be an input or an output device, but cannot be used for storage. Transfer rates for devices are always quoted in base 10, so that 10 Mbit/sec = 10,000,000 bits/sec.

I/O requests Reads or writes to I/O devices.

In other applications, we care primarily about response time, which you will recall is the total elapsed time to accomplish a particular task. If the **I/O requests** are extremely large, response time will depend heavily on bandwidth, but in many environments most accesses will be small, and the I/O system with the lowest latency per access will deliver the best response time. On single-user machines such as desktop computers and laptops, response time is the key performance characteristic.

A large number of applications, especially in the vast commercial market for computing, require both high throughput and short response times. Examples include automatic teller machines (ATMs), order entry and inventory tracking systems, file servers, and Web servers. In such environments, we care about both how long each task takes *and* how many tasks we can process in a second. The number of ATM requests you can process per hour doesn't matter if each one takes 15 minutes—you won't have any customers left! Similarly, if you can process each ATM request quickly but can only handle a small number of requests at once, you won't be able to support many ATMs, or the cost of the computer per ATM will be very high.

In summary, the three classes of desktop, server, and embedded computers are sensitive to I/O dependability and cost. Desktop and embedded systems are more

focused on response time and diversity of I/O devices, while server systems are more focused on throughput and expandability of I/O devices.

8.2 Disk Storage and Dependability

As mentioned in Chapter 1, magnetic disks rely on a rotating platter coated with a magnetic surface and use a moveable read/write head to access the disk. Disk storage is **nonvolatile**—the data remains even when power is removed. A magnetic disk consists of a collection of platters (1–4), each of which has two recordable disk surfaces. The stack of platters is rotated at 5400 to 15,000 RPM and has a diameter from an inch to just over 3.5 inches. Each disk surface is divided into concentric circles, called **tracks**. There are typically 10,000 to 50,000 tracks per surface. Each track is in turn divided into **sectors** that contain the information; each track may have 100 to 500 sectors. Sectors are typically 512 bytes in size, although there is an initiative to increase the sector size to 4096 bytes. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code (see  **Appendix B**, page B-64), a gap, the sector number of the next sector, and so on. Originally, all tracks had the same number of sectors and hence the same number of bits, but with the introduction of zone bit recording (ZBR) in the early 1990s, disk drives changed to a varying number of sectors (and hence bits) per track, instead keeping the spacing between bits constant. ZBR increases the number of bits on the outer tracks and thus increases the drive capacity.

nonvolatile Storage device where data retains its value even when power is removed.

track One of thousands of concentric circles that makes up the surface of a magnetic disk.

sector One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk.

As we saw in Chapter 1, to read and write information the read/write heads must be moved so that they are over the correct location. The disk heads for each surface are connected together and move in conjunction, so that every head is over the same track of every surface. The term *cylinder* is used to refer to all the tracks under the heads at a given point on all surfaces.

To access data, the operating system must direct the disk through a three-stage process. The first step is to position the head over the proper track. This operation is called a **seek**, and the time to move the head to the desired track is called the *seek time*.

seek The process of positioning a read/write head over the proper track on a disk.

Disk manufacturers report minimum seek time, maximum seek time, and average seek time in their manuals. The first two are easy to measure, but the average is open to wide interpretation because it depends on the seek distance. The industry has decided to calculate average seek time as the sum of the time for all possible seeks divided by the number of possible seeks. Average seek times are usually advertised as 3 ms to 14 ms, but, depending on the application and scheduling of disk requests, the actual average seek time may be only 25% to 33% of the

advertised number because of locality of disk references. This locality arises both because of successive accesses to the same file and because the operating system tries to schedule such accesses together.

rotation latency Also called delay. The time required for the desired sector of a disk to rotate under the read/write head; usually assumed to be half the rotation time.

Once the head has reached the correct track, we must wait for the desired sector to rotate under the read/write head. This time is called the **rotational latency** or **rotational delay**. The average latency to the desired information is halfway around the disk. Because the disks rotate at 5400 RPM to 15,000 RPM, the average rotational latency is between

$$\text{Average rotational latency} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM}} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM} \left/ 60 \frac{\text{seconds}}{\text{minute}} \right.}$$

$$= 0.0056 \text{ seconds} = 5.6 \text{ ms}$$

and

$$\text{Average rotational latency} = \frac{0.5 \text{ rotation}}{15{,}000 \text{ RPM}} = \frac{0.5 \text{ rotation}}{15{,}000 \text{ RPM} \left/ 60 \frac{\text{seconds}}{\text{minute}} \right.}$$

$$= 0.0020 \text{ seconds} = 2.0 \text{ ms}$$

The last component of a disk access, *transfer time,* is the time to transfer a block of bits. The transfer time is a function of the sector size, the rotation speed, and the recording density of a track. Transfer rates in 2004 are between 30 and 80 MB/sec. The one complication is that most disk controllers have a built-in cache that stores sectors as they are passed over; transfer rates from the cache are typically higher and may be up to 320 MB/sec in 2004. Today, most disk transfers are multiple sectors in length.

A *disk controller* usually handles the detailed control of the disk and the transfer between the disk and the memory. The controller adds the final component of disk access time, *controller time*, which is the overhead the controller imposes in performing an I/O access. The average time to perform an I/O operation will consist of these four times plus any wait time incurred because other processes are using the disk.

EXAMPLE

Disk Read Time

What is the average time to read or write a 512-byte sector for a typical disk rotating at 10,000 RPM? The advertised average seek time is 6 ms, the transfer rate is 50 MB/sec, and the controller overhead is 0.2 ms. Assume that the disk is idle so that there is no waiting time.

ANSWER

Average disk access time is equal to Average seek time + Average rotational delay + Transfer time + Controller overhead. Using the advertised average seek time, the answer is

$$6.0 \text{ ms} + \frac{0.5 \text{ rotation}}{10{,}000 \text{ RPM}} + \frac{0.5 \text{ KB}}{50 \text{ MB/sec}} + 0.2 \text{ ms} = 6.0 + 3.0 + 0.01 + 0.2 = 9.2 \text{ ms}$$

If the measured average seek time is 25% of the advertised average time, the answer is

$$1.5 \text{ ms} + 3.0 \text{ ms} + 0.01 \text{ ms} + 0.2 \text{ ms} = 4.7 \text{ ms}$$

Notice that when we consider measured average seek time, as opposed to advertised average seek time, the rotational latency can be the largest component of the access time.

Disk densities have continued to increase for more than 50 years. The impact of this compounded improvement in density and the reduction in physical size of a disk drive has been amazing, as Figure 8.3 shows. The aims of different disk designers have led to a wide variety of drives being available at any particular time. Figure 8.4 shows the characteristics of three magnetic disks. In 2004, these disks from a single manufacturer cost between $0.50 and $5 per gigabyte, depending on size, interface, and performance. The smaller drive has advantages in power and volume per byte.

Elaboration: Most disk controllers include caches. Such caches allow for fast access to data that was recently read between transfers requested by the CPU. They use write through and do not update on a write miss. They often also include prefetch algorithms to try to anticipate demand. Of course, such capabilities complicate the measurement of disk performance and increase the importance of workload choice.

Dependability, Reliability, and Availability

Users crave dependable storage, but how do you define it? In the computer industry, it is harder than looking it up in the dictionary. After considerable debate, the following is considered the standard definition (Laprie 1985):

> *Computer system dependability is the quality of delivered service such that reliance can justifiably be placed on this service. The service delivered by a system is its observed actual behavior as perceived by other system(s) interacting with this system's users. Each module also has an ideal specified behavior, where a service specification is an agreed description of the expected behavior. A system failure occurs when the actual behavior deviates from the specified behavior.*

- The OS handles the interrupts generated by I/O devices, just as it handles the exceptions generated by a program.
- The OS tries to provide equitable access to the shared I/O resources, as well as schedule accesses in order to enhance system throughput.

To perform these functions on behalf of user programs, the operating system must be able to communicate with the I/O devices and to prevent the user program from communicating with the I/O devices directly. Three types of communication are required:

1. The OS must be able to give commands to the I/O devices. These commands include not only operations like read and write, but also other operations to be done on the device, such as a disk seek.
2. The device must be able to notify the OS when the I/O device has completed an operation or has encountered an error. For example, when a disk completes a seek, it will notify the OS.
3. Data must be transferred between memory and an I/O device. For example, the block being read on a disk read must be moved from disk to memory.

In the next few sections, we will see how these communications are performed.

Giving Commands to I/O Devices

To give a command to an I/O device, the processor must be able to address the device and to supply one or more command words. Two methods are used to address the device: memory-mapped I/O and special I/O instructions. In **memory-mapped I/O**, portions of the address space are assigned to I/O devices. Reads and writes to those addresses are interpreted as commands to the I/O device.

memory-mapped I/O An I/O scheme in which portions of address space are assigned to I/O devices and reads and writes to those addresses are interpreted as commands to the I/O device.

For example, a write operation can be used to send data to an I/O device where the data will be interpreted as a command. When the processor places the address and data on the memory bus, the memory system ignores the operation because the address indicates a portion of the memory space used for I/O. The device controller, however, sees the operation, records the data, and transmits it to the device as a command. User programs are prevented from issuing I/O operations directly because the OS does not provide access to the address space assigned to the I/O devices and thus the addresses are protected by the address translation. Memory-mapped I/O can also be used to transmit data by writing or reading to select addresses. The device uses the address to determine the type of command, and the data may be provided by a write or obtained by a read. In any event, the address encodes both the device identity and the type of transmission between processor and device.

Actually performing a read or write of data to fulfill a program request usually requires several separate I/O operations. Furthermore, the processor may have to interrogate the status of the device between individual commands to determine whether the command completed successfully. For example, a simple printer has two I/O device registers—one for status information and one for data to be printed. The Status register contains a *done bit*, set by the printer when it has printed a character, and an *error bit*, indicating that the printer is jammed or out of paper. Each byte of data to be printed is put into the Data register. The processor must then wait until the printer sets the done bit before it can place another character in the buffer. The processor must also check the error bit to determine if a problem has occurred. Each of these operations requires a separate I/O device access.

I/O instructions A dedicated instruction that is used to give a command to an I/O device and that specifies both the device number and the command word (or the location of the command word in memory).

Elaboration: The alternative to memory-mapped I/O is to use dedicated I/O instructions in the processor. These I/O instructions can specify both the device number and the command word (or the location of the command word in memory). The processor communicates the device address via a set of wires normally included as part of the I/O bus. The actual command can be transmitted over the data lines in the bus. Examples of computers with I/O instructions are the Intel IA-32 and the IBM 370 computers. By making the I/O instructions illegal to execute when not in kernel or supervisor mode, user programs can be prevented from accessing the devices directly.

Communicating with the Processor

polling The process of periodically checking the status of an I/O device to determine the need to service the device.

The process of periodically checking status bits to see if it is time for the next I/O operation, as in the previous example, is called **polling**. Polling is the simplest way for an I/O device to communicate with the processor. The I/O device simply puts the information in a Status register, and the processor must come and get the information. The processor is totally in control and does all the work.

Polling can be used in several different ways. Real-time embedded applications poll the I/O devices since the I/O rates are predetermined and it makes I/O overhead more predictable, which is helpful for real time. As we will see, this allows polling to be used even when the I/O rate is somewhat higher.

The disadvantage of polling is that it can waste a lot of processor time because processors are so much faster than I/O devices. The processor may read the Status register many times, only to find that the device has not yet completed a comparatively slow I/O operation, or that the mouse has not budged since the last time it was polled. When the device completes an operation, we must still read the status to determine whether it was successful.

interrupt-driven I/O An I/O scheme that employs interrupts to indicate to the processor that an I/O device needs attention.

The overhead in a polling interface was recognized long ago, leading to the invention of interrupts to notify the processor when an I/O device requires attention from the processor. **Interrupt-driven I/O**, which is used by almost all systems

for at least some devices, employs I/O interrupts to indicate to the processor that an I/O device needs attention. When a device wants to notify the processor that it has completed some operation or needs attention, it causes the processor to be interrupted.

An I/O interrupt is just like the exceptions we saw in Chapters 5, 6, and 7, with two important exceptions:

1. An I/O interrupt is asynchronous with respect to the instruction execution. That is, the interrupt is not associated with any instruction and does not prevent the instruction completion. This is very different from either page fault exceptions or exceptions such as arithmetic overflow. Our control unit need only check for a pending I/O interrupt at the time it starts a new instruction.
2. In addition to the fact that an I/O interrupt has occurred, we would like to convey further information such as the identity of the device generating the interrupt. Furthermore, the interrupts represent devices that may have different priorities and whose interrupt requests have different urgencies associated with them.

To communicate information to the processor, such as the identity of the device raising the interrupt, a system can use either vectored interrupts or an exception Cause register. When the processor recognizes the interrupt, the device can send either the vector address or a status field to place in the Cause register. As a result, when the OS gets control, it knows the identity of the device that caused the interrupt and can immediately interrogate the device. An interrupt mechanism eliminates the need for the processor to poll the device and instead allows the processor to focus on executing programs.

Interrupt Priority Levels

To deal with the different priorities of the I/O devices, most interrupt mechanisms have several levels of priority: UNIX operating systems use four to six levels. These priorities indicate the order in which the processor should process interrupts. Both internally generated exceptions and external I/O interrupts have priorities; typically, I/O interrupts have lower priority than internal exceptions. There may be multiple I/O interrupt priorities, with high-speed devices associated with the higher priorities.

To support priority levels for interrupts, MIPS provides the primitives that let the operating system implement the policy, similar to how MIPS handles TLB misses. Figure 8.13 shows the key registers, and Section A.7 in **Appendix A** gives more details.

The Status register determines who can interrupt the computer. If the interrupt enable bit is 0, then none can interrupt. A more refined blocking of interrupts is available in the interrupt mask field. There is a bit in the mask corresponding to

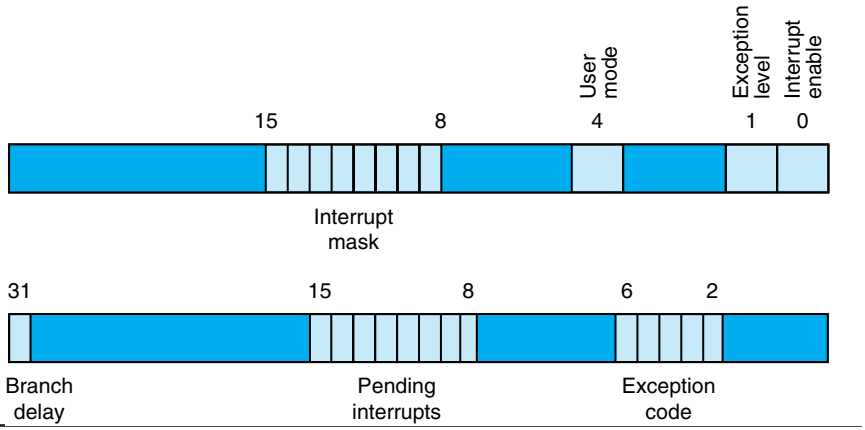


FIGURE 8.13 The Cause and Status registers. This version of the Cause register corresponds to the MIPS-32 architecture. The earlier MIPS I architecture had three nested sets of kernel/user and interrupt enable bits to support nested interrupts. Section A.7 in Appendix A has more detials about these registers.

each bit in the pending interrupt field of the Cause register. To enable the corresponding interrupt, there must be a 1 in the mask field at that bit position. Once an interrupt occurs, the operating system can find the reason in the exception code field of the Status register: 0 means an interrupt occurred, with other values for the exceptions mentioned in Chapter 7.

Here are the steps that must occur in handling an interrupt:

1. Logically AND the pending interrupt field and the interrupt mask field to see which enabled interrupts could be the culprit. Copies are made of these two registers using the `mfc0` instruction.
2. Select the higher priority of these interrupts. The software convention is that the leftmost is the highest priority.
3. Save the interrupt mask field of the Status register.
4. Change the interrupt mask field to disable all interrupts of equal or lower priority.
5. Save the processor state needed to handle the interrupt.
6. To allow higher-priority interrupts, set the interrupt enable bit of the Cause register to 1.
7. Call the appropriate interrupt routine.
8. Before restoring state, set the interrupt enable bit of the Cause register to 0. This allows you to restore the interrupt mask field.

Appendix A shows an exception handler for a simple I/O task on pages A-36 to A-37.

How do the *interrupt priority levels* (IPL) correspond to these mechanisms? The IPL is an operating system invention. It is stored in the memory of the process, and every process is given an IPL. At the lowest IPL, all interrupts are permitted. Conversely, at the highest IPL, all interrupts are blocked. Raising and lowering the IPL involves changes to the interrupt mask field of the Status register.

Elaboration: The two least significant bits of the pending interrupt and interrupt mask fields are for software interrupts, which are lower priority. These are typically used by higher-priority interrupts to leave work for lower-priority interrupts to do once the immediate reason for the interrupt is handled. Once the higher-priority interrupt is finished, the lower-priority tasks will be noticed and handled.

Transferring the Data between a Device and Memory

We have seen two different methods that enable a device to communicate with the processor. These two techniques—polling and I/O interrupts—form the basis for two methods of implementing the transfer of data between the I/O device and memory. Both these techniques work best with lower-bandwidth devices, where we are more interested in reducing the cost of the device controller and interface than in providing a high-bandwidth transfer. Both polling and interrupt-driven transfers put the burden of moving data and managing the transfer on the processor. After looking at these two schemes, we will examine a scheme more suitable for higher-performance devices or collections of devices.

We can use the processor to transfer data between a device and memory based on polling. In real-time applications, the processor loads data from I/O device registers and stores them into memory.

An alternative mechanism is to make the transfer of data interrupt driven. In this case, the OS would still transfer data in small numbers of bytes from or to the device. But because the I/O operation is interrupt driven, the OS simply works on other tasks while data is being read from or written to the device. When the OS recognizes an interrupt from the device, it reads the status to check for errors. If there are none, the OS can supply the next piece of data, for example, by a sequence of memory-mapped writes. When the last byte of an I/O request has been transmitted and the I/O operation is completed, the OS can inform the program. The processor and OS do all the work in this process, accessing the device and memory for each data item transferred.

Interrupt-driven I/O relieves the processor from having to wait for every I/O event, although if we used this method for transferring data from or to a hard disk, the overhead could still be intolerable, since it could consume a large fraction of the processor when the disk was transferring. For high-bandwidth devices like hard disks, the transfers consist primarily of relatively large blocks of data

(hundreds to thousands of bytes). Thus, computer designers invented a mechanism for offloading the processor and having the device controller transfer data directly to or from the memory without involving the processor. This mechanism is called **direct memory access** (DMA). The interrupt mechanism is still used by the device to communicate with the processor, but only on completion of the I/O transfer or when an error occurs.

direct memory access (DMA) A mechanism that provides a device controller the ability to transfer data directly to or from the memory without involving the processor.

bus master A unit on the bus that can initiate bus requests.

DMA is implemented with a specialized controller that transfers data between an I/O device and memory independent of the processor. The DMA controller becomes the **bus master** and directs the reads or writes between itself and memory. There are three steps in a DMA transfer:

1. The processor sets up the DMA by supplying the identity of the device, the operation to perform on the device, the memory address that is the source or destination of the data to be transferred, and the number of bytes to transfer.
2. The DMA starts the operation on the device and arbitrates for the bus. When the data is available (from the device or memory), it transfers the data. The DMA device supplies the memory address for the read or the write. If the request requires more than one transfer on the bus, the DMA unit generates the next memory address and initiates the next transfer. Using this mechanism, a DMA unit can complete an entire transfer, which may be thousands of bytes in length, without bothering the processor. Many DMA controllers contain some memory to allow them to deal flexibly with delays either in transfer or those incurred while waiting to become bus master.
3. Once the DMA transfer is complete, the controller interrupts the processor, which can then determine by interrogating the DMA device or examining memory whether the entire operation completed successfully.

There may be multiple DMA devices in a computer system. For example, in a system with a single processor-memory bus and multiple I/O buses, each I/O bus controller will often contain a DMA processor that handles any transfers between a device on the I/O bus and the memory.

Unlike either polling or interrupt-driven I/O, DMA can be used to interface a hard disk without consuming all the processor cycles for a single I/O. Of course, if the processor is also contending for memory, it will be delayed when the memory is busy doing a DMA transfer. By using caches, the processor can avoid having to access memory most of the time, thereby leaving most of the memory bandwidth free for use by I/O devices.

Elaboration: To further reduce the need to interrupt the processor and occupy it in handling an I/O request that may involve doing several actual operations, the I/O controller can be made more intelligent. Intelligent controllers are often called *I/O proces-*

sors (as well as *I/O controllers* or *channel controllers*). These specialized processors basically execute a series of I/O operations, called an *I/O program*. The program may be stored in the I/O processor, or it may be stored in memory and fetched by the I/O processor. When using an I/O processor, the operating system typically sets up an I/O program that indicates the I/O operations to be done as well as the size and transfer address for any reads or writes. The I/O processor then takes the operations from the I/O program and interrupts the processor only when the entire program is completed. DMA processors are essentially special-purpose processors (usually single-chip and nonprogrammable), while I/O processors are often implemented with general-purpose microprocessors, which run a specialized I/O program.

Direct Memory Access and the Memory System

When DMA is incorporated into an I/O system, the relationship between the memory system and processor changes. Without DMA, all accesses to the memory system come from the processor and thus proceed through address translation and cache access as if the processor generated the references. With DMA, there is another path to the memory system—one that does not go through the address translation mechanism or the cache hierarchy. This difference generates some problems in both virtual memory systems and systems with caches. These problems are usually solved with a combination of hardware techniques and software support.

The difficulties in having DMA in a virtual memory system arise because pages have both a physical and a virtual address. DMA also creates problems for systems with caches because there can be two copies of a data item: one in the cache and one in memory. Because the DMA processor issues memory requests directly to the memory rather than through the processor cache, the value of a memory location seen by the DMA unit and the processor may differ. Consider a read from disk that the DMA unit places directly into memory. If some of the locations into which the DMA writes are in the cache, the processor will receive the old value when it does a read. Similarly, if the cache is write-back, the DMA may read a value directly from memory when a newer value is in the cache, and the value has not been written back. This is called the *stale data problem* or *coherence problem*.

Hardware Software Interface

In a system with virtual memory, should DMA work with virtual addresses or physical addresses? The obvious problem with virtual addresses is that the DMA unit will need to translate the virtual addresses to physical addresses. The major problem with the use of a physical address in a DMA transfer is that the transfer cannot easily cross a page boundary. If an I/O request crossed a page boundary, then the memory locations to which it was being transferred would not necessarily be contiguous in the virtual memory. Consequently, if we use physical addresses, we must constrain all DMA transfers to stay within one page.