Chapter 10 Input/Output and Stable Storage
(Revision number 10)

The next step in demystifying what is inside a box is understanding the I/O subsystem. In Chapter 4, we discussed interrupts, a mechanism for I/O devices to grab the attention of the processor. In this chapter, we will present more details on the interaction between processor and I/O devices, as well as different methods of transferring data among the processor, memory, and I/O devices.

We start out by discussing the basic paradigms for the CPU to communicate with I/O devices. We then identify the hardware mechanisms needed to carry out such communication in hardware, as well as the details of the buses that serve as the conduits for transferring data between the CPU and I/O. Complementary to the hardware mechanisms is the operating system entity, called a device driver that carries out the actual dialogue between the CPU and the I/O devices. Stable storage, often referred to as the *hard drive*, is unquestionably one of the most important and intricate member of the I/O devices found inside a box. We discuss the details of the hard disk including scheduling algorithms for orchestrating the I/O activities.

10.1 Communication between the CPU and the I/O devices

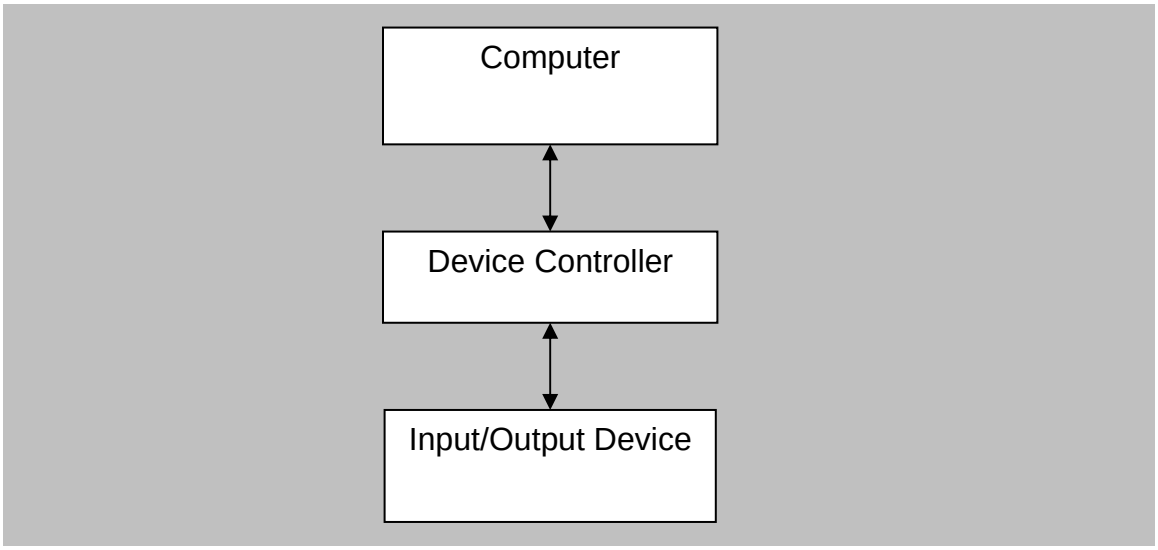


Figure 10.1 Relationship of device controller to other components

Although we started our exploration of the computer system with the processor, most users of computers may not even be aware of the processor that is inside the gadget they are using. Take for example the cellphone or the iPod. It is the functionality provided by iPod that attracts a teenager (or even adults for that matter) to it. An iPod or a cellphone get its utility from the input/output devices that each provides for interacting with it. Thus, knowing how I/O devices interact with the rest of the computer system is a key component to unraveling the "box." Although there are a wide variety of I/O devices, their connection to the rest of the system is quite similar. As we have seen so far, the

processor executes instructions in its instruction set repertoire. LC-2200 has no special instruction that would make it communicate with a CD player or a speaker directly. Let's understand how a device such an iPod plays music.

A special piece of hardware known as a *device controller* acts as an intermediary between an I/O device and the computer. This controller knows how to communicate with both the I/O device and with the computer.

10.1.1 Device controller

To make this discussion concrete, let us consider a very simple device, the *keyboard*. The device itself has circuitry inside it to map the mechanical action of tapping on a key to a binary encoding of the character that the key represents. This binary encoding, usually in a format called ASCII (*American Standard Code for Information Interchange)*, has to be conveyed to the computer. For this information exchange to happen, two things are necessary. First, we need temporary space to hold the character that was typed. Second, we have to grab the attention of the processor to give it this character. This is where the device controller comes in. Let us consider the minimal smarts needed in the keyboard device controller. It has two registers: *data* register, and *status* register. The data register is the storage space for the character typed on the keyboard. As the name suggests, the status register is an aggregation of the current state of information exchange of the device with the computer.

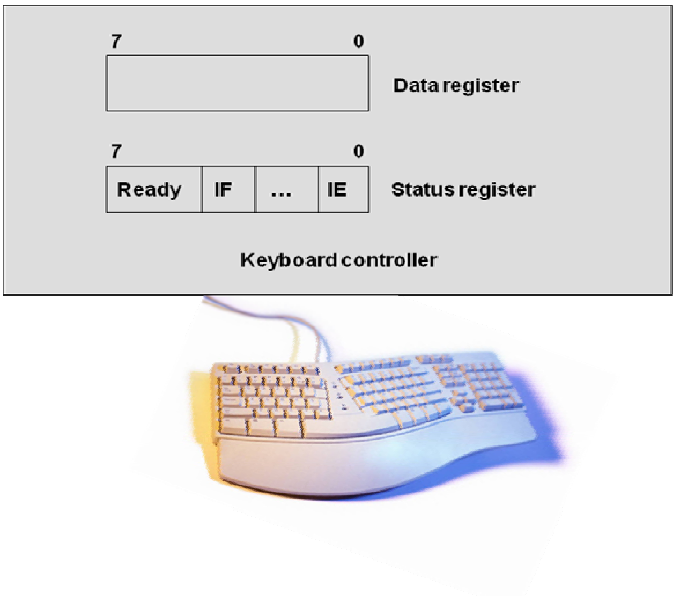


Figure 10.2: Keyboard controller

With respect to the keyboard, the state includes:

- A bit, called *ready* bit that represents the answer to the question, "is the character in the data register new (i.e. not seen by the processor yet)?"
- A bit, called *interrupt enable (IE)* that represents the answer to the question, "is the processor allowing the device to interrupt it?"

- A bit, called *interrupt flag (IF)* that serves to answer the question, "is the controller ready to interrupt the processor?

Figure 10.2 shows such a minimal keyboard controller. Depending on the sophistication of the device, additional status information may need to be included. For example, if the input rate of the device exceeds the rate of information transfer to the processor, a *data overrun flag* may record this status on the controller.

Sometimes it may be a good idea to separate the *status* that comes from the device, and the *command* that comes from the processor. The keyboard is a simple enough device that the only command from the processor is to turn on or off the interrupt enable bit. A more sophisticated device (for example a camera) may require additional commands (take picture, pan, tilt, etc.). In general, a device controller may have a set of registers through which it interacts with the processor.

10.1.2 Memory Mapped I/O

Next, we investigate how to connect the device controller to the computer. Somehow, we have to make the registers in the controller visible to the processor. A simple way to accomplish this is via the processor-memory bus as shown in Figure 10.3. The processor reads and writes to memory using load/store instructions. If the registers in the controller (data and status) appear as memory locations to the CPU then it can simply use the same load/store instructions to manipulate these registers. This technique, referred to as *memory mapped I/O*, allows interaction between the processor and the device controller without any change to the processor.

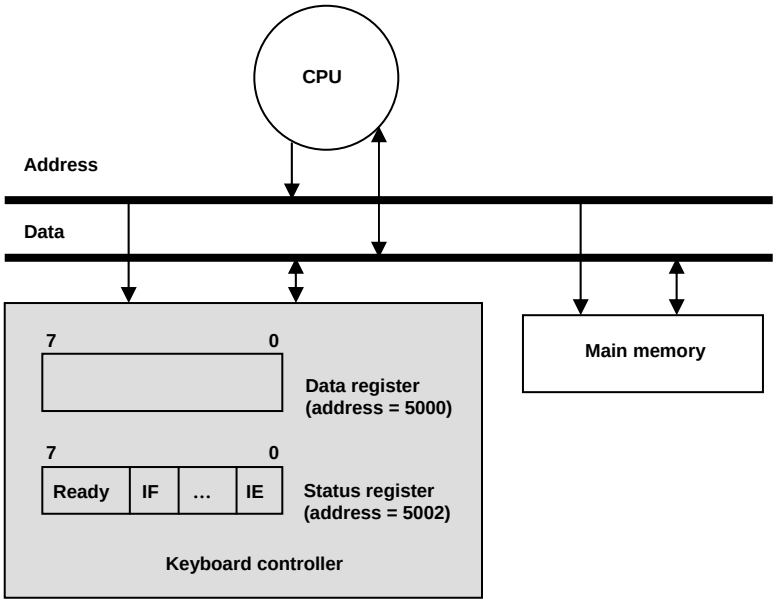


Figure 10.3: Connecting the device controller to the processor-memory bus

It is a real simple trick to make the device registers appear as memory locations. We give the device registers unique memory addresses. For example, let us arbitrarily give the

data register the memory address 5000 and the status register the memory address 5002. The keyboard controller has smarts (i.e. circuitry) in it to react to these two addresses appearing on the address bus. For example, if the processor executes the instruction:

```
Load r1, Mem[5000]
```

The controller recognizes that address 5000 corresponds to the data register, and puts the contents of the data register on the data bus. The processor has no idea that it is coming from a special register on the controller. It simply takes the value appearing on the data bus and stores it into register **`r1`**.

This is the power of memory mapped I/O. Without adding any new instructions to the instruction set, we have now integrated the device controller into the computer system. You may be wondering if this technique causes confusion between the memory that also reacts to addresses appearing on the bus and the device controllers. The basic idea is to reserve a portion of the address space for such device controllers. Assume that we have a 32-bit processor and that we wish to reserve 64Kbytes for I/O device addresses. We could arbitrarily designate addresses in the range `0xFFFF0000` through `0xFFFFFFFF` for I/O devices, and assign addresses within this range to the devices we wish to connect to the bus. For example, we may assign `0xFFFF0000` as the data register and `0xFFFF0002` as the status register for the keyboard controller. The design of the keyboard device controller considers this assignment and reacts to these addresses. What this means is that every device controller has circuitry to decode an address appearing on the bus. If an address appearing on the bus corresponds to one assigned to this controller, then it behaves like "memory" for the associated command on the bus (read or write). Correspondingly, the design of the memory system will be to ignore addresses in the range designated for I/O. Of course, it is a matter of convention as to the range of addresses reserved for I/O. The usual convention is to reserve high addresses for I/O device registers.

Some of you are perhaps wondering how to reconcile this discussion with the details of the memory hierarchy that we just finished learning about in Chapter 9. If a memory address assigned to a device register were present in the cache, wouldn't the processor get a stale value from the cache, instead of the contents of the device register? This is a genuine concern and it is precisely for this reason that cache controllers provide the ability to treat certain regions of memory as "uncachable". Essentially, even though the processor reads a device register as if it were a memory location, the cache controller has been set *a priori* to not encache such locations but read them afresh every time the processor accesses them.

The advantage of memory mapped I/O is that no special I/O instructions are required but the disadvantage is that a portion of the memory address space is lost to device registers, and hence not available for users or the operating system for code and data. Some processors provide special I/O instructions and connect I/O devices to a separate I/O bus. Such designs (called *I/O mapped I/O*) are especially popular in the domain of embedded systems where memory space is limited. However, with modern general-purpose processors with a 64-bit address space, reserving a small portion of this large address space for I/O seems a reasonable design choice. Thus, memory mapped I/O is the design

of choice for modern processors, especially since it integrates easily into the processor architecture without the need for new instructions.

10.2 Programmed I/O

Now that we know how to interface a device via a device controller to the processor, let us turn our attention to moving data back and forth between a device and the processor. *Programmed I/O* (*PIO*) refers to writing a computer program that accomplishes such data transfer. To make the discussion concrete, we will use the keyboard controller we introduced earlier.

Let us summarize the actions of the keyboard controller:

1. It sets the ready bit in the status register when a new character gets into the data register.
2. Upon the CPU reading the data register, the controller clears the ready bit.

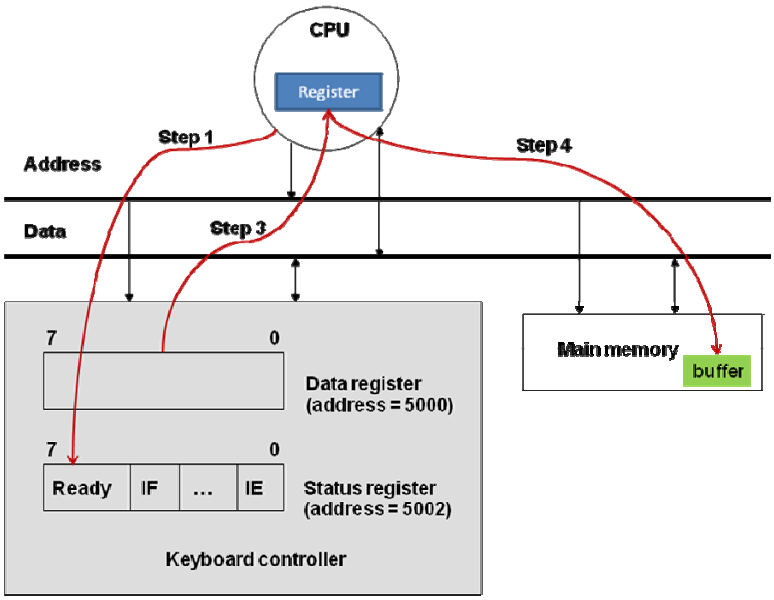


Figure 10.4: An example of PIO Data Transfer

With the above semantics of the keyboard controller, we can write a simple program to move data from the keyboard to the processor (shown pictorially in Figure 10.4):

1. Check the ready bit (step 1 in Figure 10.4).
2. If not set go to step 1.
3. Read the contents of the data register (step 3 in Figure 10.4).
4. Store the character read into memory (step 4 in Figure 10.4).
5. Go to step 1.

Step 1 and 2 constitute the handshake between the processor and the device controller. The processor continuously checks if the device has new data through these two steps. In other words, the processor *polls* the device for new data. Consider the speed differential between the processor and a device such as a keyboard. A 1 GHz processor executes an instruction in roughly 1 ns. Even if one were to type at an amazing speed, say of 300

characters a minute, the controller inputs a character only once every 200 ms. The processor would have been polling the ready bit of the status register several million times before it gets to input a character. This is an inefficient use of the processor resource. Instead of polling, the processor could enable the interrupt bit for a device and upon an interrupt execute the instructions as above to complete the data transfer. The operating system schedules other programs to run on the CPU, and context switches to handle the interrupts as we described in Chapter 4.

Programmed data transfer, accomplished either by polling or interrupt, works for slow speed devices (for example, keyboard and mouse) that typically produce data *asynchronously*. That is the data production is not rhythmic with any clock pulse. However, high-speed devices such as disks produce data *synchronously*. That is the data production follows a rhythmic clock pulse. If the processor fails to pick up the data when the device has it ready, then the device may likely overwrite it with new data, leading to data loss. The memory bus bandwidth of state-of-the-art processors is around 200Mbytes/sec. All the entities on the bus, processor and device controllers, share this bandwidth. A state-of-the-art disk drive produces data at the rate of 150Mbytes/sec. Given the limited margin between the production rate of data and available memory bus bandwidth, it is just not feasible to orchestrate the data transfer between memory and a synchronous high-speed device such as a disk through programmed I/O.

Further, even for slow speed devices using the processor to orchestrate the data transfer through programmed I/O is an inefficient use of the processor resource. In the next subsection, we introduce a different technique for accomplishing this data transfer.

10.3 DMA

Direct Memory Access (DMA), as the name suggests is a technique where the device controller has the capability to transfer data between itself and memory without the intervention of the processor.

The transfer itself is initiated by the processor, but once initiated the device carries out the transfer. Let us try to understand the smarts needed in the DMA controller. We will consider that the controller is interfacing a synchronous high-speed device such as a disk. We refer to such devices as *streaming devices*: once data transfer starts in either direction ("from" or "to" the device), data moves in or out the device continuously until completion of the transfer.

As an analogy, let us say you are cooking. You need to fill a large cooking vessel on the stove with water. You are using a small cup, catching water from the kitchen faucet, pouring it into the vessel, and going back to refilling the cup and repeating this process until the vessel is full. Now, you know that if you keep the faucet open water will continue to stream. Therefore, you use the cup as a *buffer* between the faucet that runs continuously and the vessel, turning the faucet on and off as needed.

A streaming device is like a water faucet. To read from the device, the device controller turns on the device, gets a stream of bits, turns it off, transfers the bits to memory, and

repeats the process until done with the entire data transfer. It does the reverse to write to the device. To make it concrete let us focus on reading from the device into memory. Let us see what is involved in transferring the bits to memory. The controller acquires the bus and sends one byte (or whatever the granularity of the bus transfer is) at a time to memory. Recall that the bus is a shared resource. There are other contenders for the bus including the processor. Therefore, the data transfer between the device controller and the memory is asynchronous; worse yet, the controller may not get the bus for quite a while if there are other higher priority requests for the bus. To smooth out this dichotomy of synchronous device and an asynchronous bus, the controller needs a *hardware buffer* similar to the cup in the above analogy. It is fairly intuitive to reason that the buffer should be as big as the *unit of synchronous transfer* between the device and device controller. For example, if the device is a camcorder, then the size of a single image frame (e.g., 100 K pixels) may be the smallest unit of synchronous data transfer between the device and the controller. Thus, in the case of a device controller for a camcorder, the size of the hardware buffer should be at least as big as one image frame.

To initiate a transfer, the processor needs to convey four pieces of information to the device controller: *command*, *address on the device*, *memory buffer address*, and *amount of data transfer*. Note that the data transfer is between contiguous regions of the device space and memory space. In addition, the controller has a status register to record the device status. As in the case of the keyboard controller, we can assign five memory-mapped registers in the controller: *command, status, device address, memory buffer address,* and *count*. All of these registers have unique memory addresses assigned to them. Figure 10.5 shows a simplified block diagram for a DMA controller for such a streaming device.

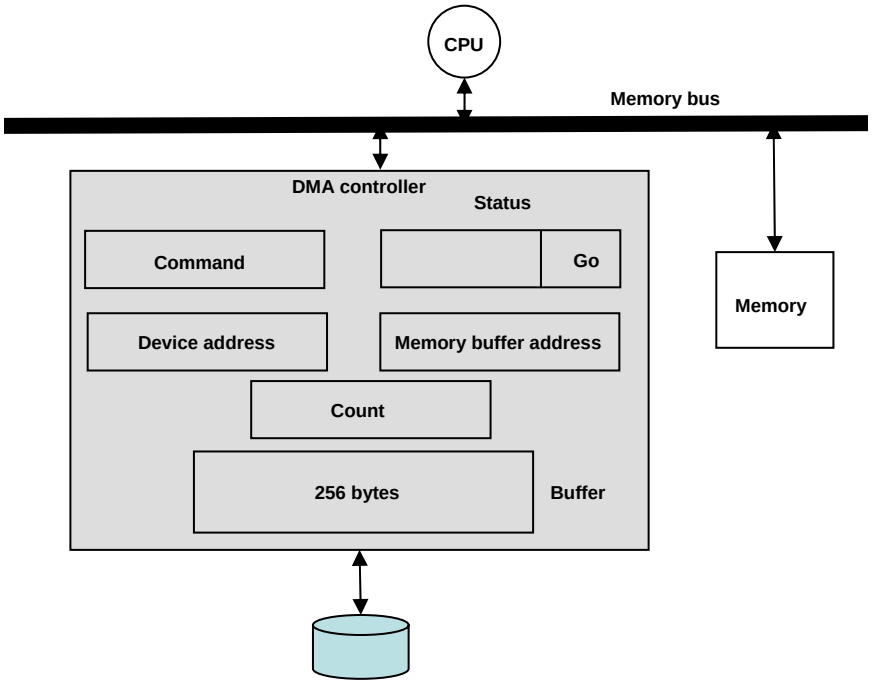


Figure 10.5: DMA controller

For example, to transfer *N bytes* of data from a memory buffer at address *M* to the device at address *D,* the CPU executes the following instructions:

1. Store *N* into the *Count* register.
2. Store *M* into the *Memory buffer address* register.
3. Store *D* into the *Device address* register.
4. Store *write to the device* command into the *Command* register.
5. Set the *Go* bit in the *Status* register.

Note that all of the above are simple *memory store* instructions (so far as the CPU is concerned) since these registers are memory mapped.

The device controller is like a CPU. Internally, it has the data path and control to implement each of a set of instructions it gets from the processor. For instance, to complete the above transfer (see Figure 10.6) the controller will access the memory bus repeatedly to move into its buffer, *N* contiguous bytes starting from the memory address *M*. Once the buffer is ready, it will initiate the transfer of the buffer contents into the device at the specified device address. If the processor had enabled interrupt for the controller, then it will interrupt the processor upon completion of the transfer.

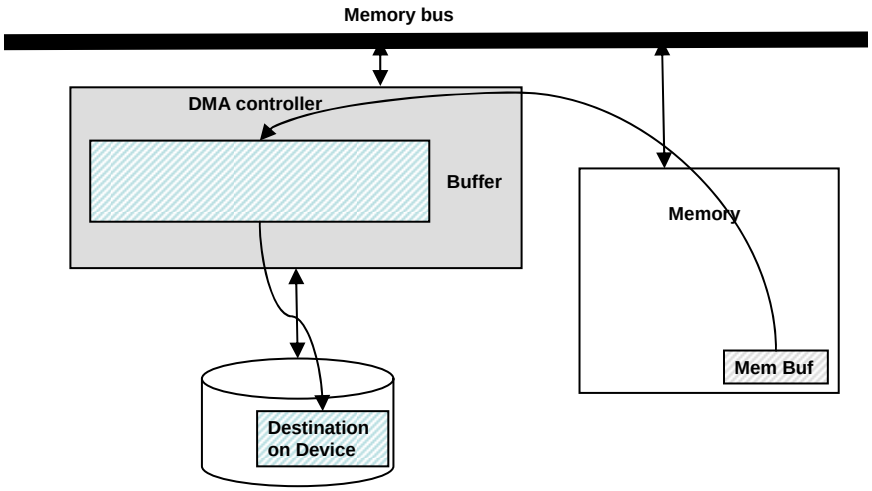


Figure 10.6: An example of DMA Data Transfer

The device controller competes with the processor for memory bus cycles, referred to as *cycle stealing*. This is an archaic term and stems from the fact that the processor is usually the bus master. The devices *steal* bus cycles when the processor does not need them. From the discussion on memory hierarchies in Chapter 9, we know that the processor mostly works out of the cache for both instructions and data. Therefore, devices stealing bus cycles from the processor does not pose a problem with a well-balanced design that accounts for the aggregate bandwidth of the memory bus being greater than the cumulative needs of the devices (including the processor) connected to it.

10.4 Buses

The system bus (or memory bus) is a key resource in the entire design. Functionally, the bus has the following components:

- Address lines
- Data lines
- Command lines
- Interrupt lines
- Interrupt acknowledge lines
- Bus arbitration lines

Electrically, high-performance systems run these wires in parallel to constitute the system bus. For example, a processor with 32-bit addressability will have 32 address lines. The number of data lines on the bus depends on the command set supported on the bus. As we have seen in the Chapter on memory hierarchy (Chapter 9), it is conceivable that the data bus is wider than the word width of the processor to support large cache block size. The command lines encode the specific command to the memory system (read, write, block read, block write, etc.), and hence should be sufficient in number to carry the binary encoding of the command set. We have seen in Chapter 4, details of the interrupt handshake between the processor and I/O devices. The number of interrupt lines (and acknowledgment lines) corresponds to the number of interrupt levels supported. In Chapter 9, we have emphasized the importance of the memory system. In a sense, the performance of the system depends crucially on the performance of the memory system. Therefore, it is essential that the system bus, the porthole for accessing the memory, be used to its fullest potential. Normally, during the current bus cycle, devices compete to acquire the bus for the next cycle. Choosing the winner for the next bus cycle happens before the current cycle completes. A variety of techniques exists for *bus arbitration*, from centralized scheme (at the processor) to more distributed schemes. Detailed discussion of bus arbitration schemes is beyond the scope of this textbook. The interested reader should refer to more advanced architecture textbooks (such as Computer Architecture: A Quantitative Approach by Hennessy and Patterson) for information on this topic.

Over the years, there have been a number of standardization efforts for buses. The purpose of standardization is to allow third party vendors to interface with the buses for developing I/O devices. Standardization poses interesting challenge to "box makers". On the one hand, such standardization helps a box maker since it increases the range of peripherals available on a given platform from third party sources. On the other hand, to keep the competitive edge, most "box makers" resist such standardization. In the real world, we will see a compromise. System buses tend to be proprietary and not adhering to any published open standards. On the other hand, I/O buses to connect peripherals tend to conform to standards. For example, *Peripheral Component Interchange (PCI)* is an open standard for I/O buses. Most box makers will support PCI buses and provide internal *bridges* to connect the PCI bus to the system bus (see Figure 10.7).

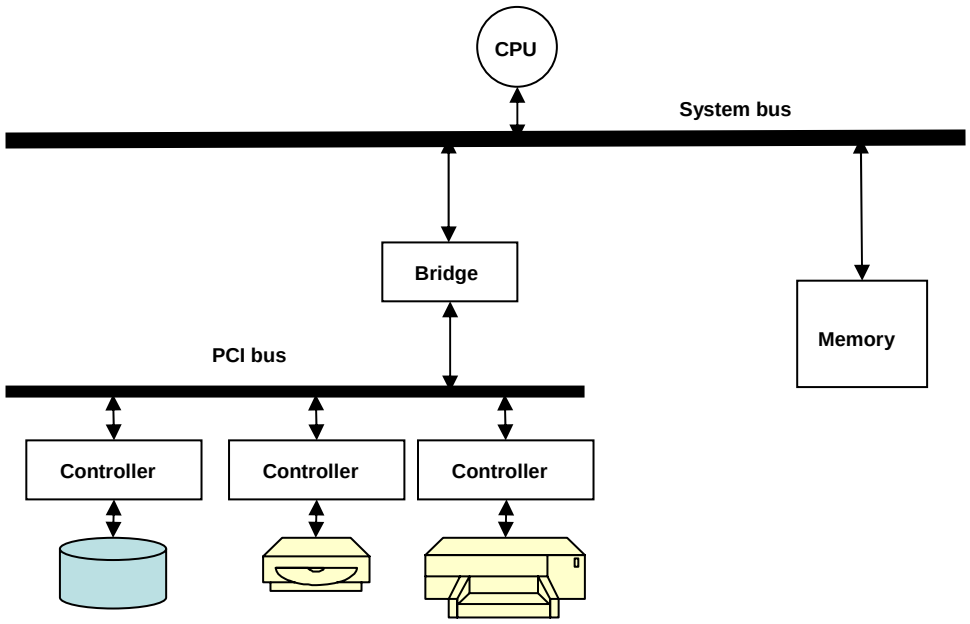


Figure 10.7: Coexistence of Standard buses and system buses

Some bus designs may electrically share the lines for multiple functions. For example, the PCI bus uses the same 32 wires for both addresses and data. The Command lines and the details of the protocol determine whether these wires carry data or addresses at any point of time.

An important consideration in the design of buses is the control regime. Buses may operate in a *synchronous* fashion. In this case, a common *bus clock* line (similar to the CPU clock) orchestrates the protocol action on individual devices. Buses may also operate in an *asynchronous* fashion. In this case, the bus *master* initiates a bus action; the bus action completes when the *slave* responds with a reply. This *master-slave* relationship obviates the need for a bus clock. To increase bus utilization, high-performance computer systems use *split transaction buses*. In this case, several independent conversations can simultaneously proceed. This complicates the design considerably in the interest of increasing the bus throughput. Such advanced topics are the purview of more advanced courses in computer architecture.

10.5 I/O Processor

In very high-performance systems used in enterprise level applications such as web servers and database servers, *I/O processors* decouple the I/O chores from the main processor. An I/O processor takes a *chain* of commands for a set of devices and carries them out without intervention from or to the main processor. The I/O processor reduces the number of interruptions that the main processor has to endure. The main processor sets up an I/O program in shared memory (see Figure 10.8), and starts up the I/O processor. The I/O processor completes the program and then interrupts the main processor.

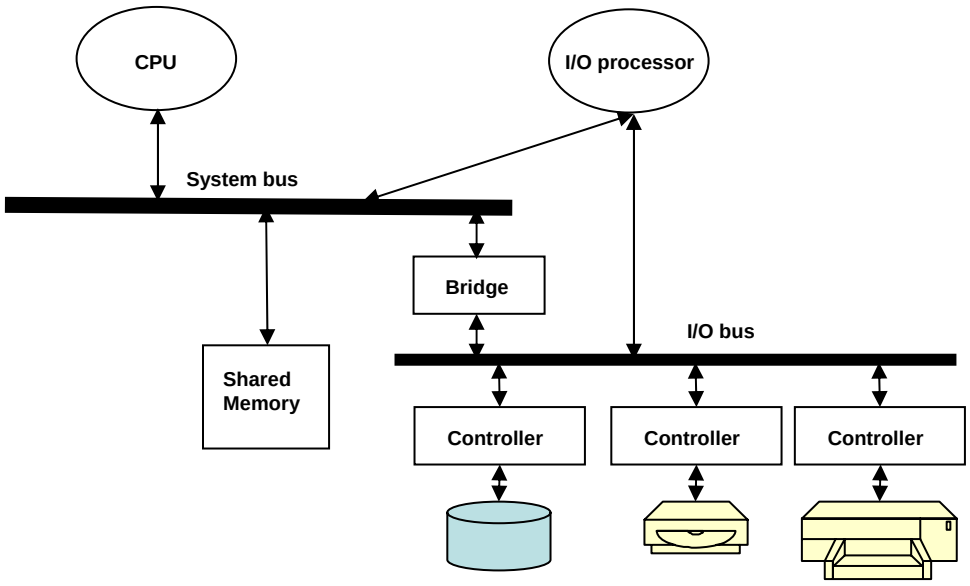


Figure 10.8: I/O processor

IBM popularized I/O processors in the era of mainframes. As it turns out, even today mainframes are popular for large enterprise servers. IBM gave these I/O processors the name *channels*. A *multiplexer channel* controls slow speed character-oriented devices such as a bank of terminals or displays. A *block multiplexer channel* controls several medium speed block-oriented devices (stream-oriented devices as we refer to them in this chapter) such as tape drives. A *selector channel* services a single high-speed stream-oriented device such as a disk.

An I/O processor is functionally similar to a DMA controller. However, it is at a higher level since it can execute a series of commands (via an I/O program) from the CPU. On the other hand, the DMA controller works in *slave* mode handling one command at a time from the CPU.

10.6 Device Driver

Device driver is a part of the operating system, and there is a device driver for controlling each device in the computer system. Figure 10.9 shows the structure of the system software, specifically the relationship between the device drivers and the rest of the operating system such as the CPU scheduler, I/O scheduler and the memory manager.

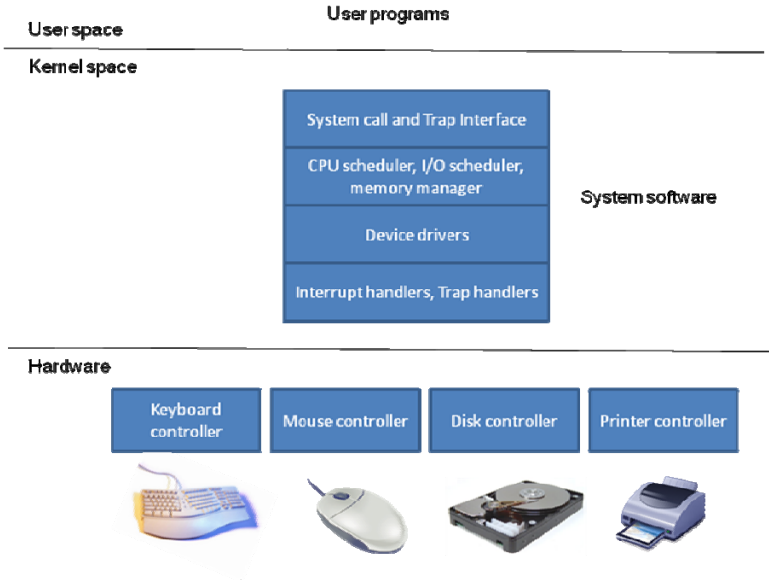


Figure 10.9: Place of the device driver in the system software stack

The specifics of the device driver software depend on the characteristics of the device it controls. For example, a keyboard driver may use interrupt-driven programmed I/O for data transfer between the keyboard controller and the CPU. On the other hand, the device driver for the hard-drive (disk) would set up the descriptors for DMA transfer between the disk controller and the memory and await an interrupt that indicates completion of the data transfer. You can see the power of abstraction at work here. It does not really matter what the details of the device is as far as the device driver is concerned. For example, the device could be some slow speed device such as a keyboard or a mouse. As far as the device driver is concerned it is simply executing code similar to the pseudo code we discussed in Section 10.2 for moving the data from the device register in the controller to the CPU. Similarly, the high-speed device may be a disk, a scanner, or a video camera. As far as the data movement is concerned, the device driver for such high-speed devices does pretty much the same thing as we discussed in the pseudo code in Section 10.3, namely, set up the descriptor for the DMA transfer and let the DMA controller associated with the device take charge. Of course, the device driver needs to take care of control functions specific to each device. As should be evident by now, there is an intimate relationship between the device controller and the device driver.

10.6.1 An Example

As a concrete example, consider a device driver for a pan-tilt-zoom (PTZ) camera. The device controller may provide a memory mapped command register for the CPU to specify the control functions such as the zoom level, tilt level, and x-y coordinates of the space in front of the camera for focusing the camera view. Similarly, the controller may provide commands to start and stop the camera. In addition to these control functions, it may implement a DMA facility for the data transfer. Table 10.1 summarizes the capabilities of the device controller for a PTZ camera.

Command	Controller Action
pan(±ɵ)	Pan the camera view by ±ɵ
tilt(±ɵ)	Tilt camera position by ±ɵ
zoom(±z)	Zoom camera focus by ±z
Start	Start camera
Stop	Stop camera
memory buffer(M)	Set memory buffer address for data transfer to M
number of frames (N)	Set number of frames to be captured and transferred to memory to N
enable interrupt	Enable interrupt from the device
disable interrupt	Disable interrupt from the device
start DMA	Start DMA data transfer from camera

Table 10.1: Summary of Commands for a PTZ Camera Controller

Device driver for such a device may consist of the following modules shown as pseudo code.

```
// device driver: camera
// The device driver performs several functions:
//   control_camera_position;
//   convey_DMA_parameters;
//   start/stop data transfer;
//   interrupt_handler;
//   error handling and reporting;
// Control camera position
camera_position_control
     (angle pan_angle; angle tilt_angle; int z)
{
     pan(pan_angle);
     tilt(tilt_angle);
     zoom(z);
}

// Set up DMA parameters for data transfer
camera_DMA_parameters(address mem_buffer;int num_frames)
{
     memory_buffer(mem_buffer);
     capture_frames(num_frames);
}
```

```
// Start DMA transfer
camera_start_data_transfer()
{
     start_camera();
     start_DMA();
}

// Stop DMA transfer
camera_stop_data_transfer();
{
     // automatically aborts data transfer
     // if camera is stopped;
     stop_camera();
}

// Enable interrupts from the device
camera_enable_interrupt()
{
     enable_interrupt();
}

// Disable interrupts from the device
camera_disable_interrupt()
{
     disable_interrupt();
}

// Device interrupt handler
camera_interrupt_handler()
{
     // This will be coded similar to any
     // interrupt handler we have seen in
     // chapter 4.
     //
     // The upshot of interrupt handling may
     // to deliver "events" to the upper layers
     // of the system software (see Figure 10.9)
     // which may be one of the following:
     //       - normal I/O request completion
     //       - device errors for the I/O request
     //

}
```

This simplistic treatment of a device driver is meant to give you the confidence that writing such a piece of software is a straightforward exercise similar to any programming assignment. We should point out that modern devices might be much more

sophisticated. For example, a modern PTZ camera may incorporate the device controller in the camera itself so that the level of interface presented to the computer is much higher. Similarly, the camera may plug directly into the local area network (we cover more on networking in Chapter 13), so communicating with the device may be similar to communicating with a peer computer on a local area network using the network protocol stack.

The main point to take away from this pseudo-code is that the code for a device driver is straightforward. From your programming experience, you already know that writing any program required taking care of corner cases (such as checking array bounds) and dealing with exceptions (such as checking the return codes on system calls). Device driver code is no different. What makes device driver code more interesting or more challenging depending on your point of view is that a number of things could happen that may have nothing to do with the logic of the device driver code. It is possible to plan for some of these situations in the device driver code. Examples include:

1. The parameters specified in the command to the controller are illegal (such as illegal values for pan, tilt, zoom, and illegal memory address for data transfer).
2. The device is already in use by some other program.
3. The device is not responding due to some reason (e.g., device is not powered on; device is malfunctioning; etc.).

Some of the situations may be totally unplanned for and could occur simply because of "human in the loop." Examples include:

1. The device is unplugged from the computer while data transfer is going on.
2. The power chord for the device is unplugged while the data transfer is going on.
3. The device starts malfunctioning during data transfer (e.g., someone accidentally knocked the camera off its moorings; etc.)

10.7 Peripheral Devices

Historically, I/O devices have been grouped into *character-oriented*[1] and *block-oriented* devices. Dot matrix printers, cathode ray terminals (CRT), teletypewriters are examples of the former. The input/output from these devices happen a character at a time. Due to the relative slowness of these devices, programmed I/O (PIO) is a viable method of data transfer to/from such devices from/to the computer system. Hard drive (disk), CD-RW, and tape player are examples of block-oriented devices. As the name suggests, such devices move a block of data to/from the device from/to the computer system. For example, with a laser printer, once it starts printing a page, it continually needs the data for that page, since there is no way to pause the laser printer in the middle of printing a page. The same is true of a magnetic tape drive. It reads or writes a block of data at a time from the magnetic tape. Data transfers from such devices are subject to the data overrun problem that we mentioned earlier in Section 10.1.1. Therefore, DMA is the only viable way of effecting data transfers between such devices and the computer system.

[1] We introduced the terms character-oriented and block-oriented earlier in Section 10.5 without formally defining these terms.

Device	Input/output	Human in the loop	Data rate (circa 2008)	PIO	DMA
Keyboard	Input	Yes	5-10 bytes/sec	X	
Mouse	Input	Yes	80-200 bytes/sec	X	
Graphics display	Output	No	200-350 MB/sec		X
Disk (hard drive)	Input/Output	No	100-200 MB/sec		X
Network (LAN)	Input/Output	No	1 Gbit/sec		X
Modem[2]	Input/Output	No	1-8 Mbit/sec		X
Inkjet printer[3]	Output	No	20-40 KB/sec	X[4]	X
Laser printer[5]	Output	No	200-400 KB/sec		X
Voice (microphone/speaker)[6]	Input/Output	Yes	10 bytes/sec	X	
Audio (music)	Output	No	4-500 KB/sec		X
Flash memory	Input/Output	No	10-50 MB/sec		X
CD-RW	Input/Output	No	10-20 MB/sec		X
DVD-R	Input	No	10-20 MB/sec		X

Table 10.2: A Snapshot of Data Rates of Computer Peripherals[7]

Table 10.2 summarizes the devices typically found in a modern computer system, their data rates (circa 2008), and the efficacy of programmed I/O versus DMA. The second column signifies whether a "human in the loop" influences the data rate from the device. Devices such as a keyboard and a mouse work at human speeds. For example, a typical typewriting speed of 300-600 characters/minute translates to a keyboard input rate of 5-10 bytes/second. Similarly, moving a mouse generating 10-20 events/second translates to an input rate of 80 to 200 bytes/second. A processor could easily handle such data rates without data loss using programmed I/O (with either polling or interrupt). A graphics display found in most modern computer systems has a frame buffer in the device controller that is updated by the device driver using DMA transfer. For a graphics display, a screen resolution of 1600 x 1200, a screen refresh rate of 60 Hz, and 24 bits per pixel, yields a data rate of over 300 MB/sec. A music CD holds over 600 MB of data with a playtime of 1 hour. A movie DVD holds over 4GB of data with a playtime of 2 hours. Both these technologies allow faster than real time reading of data of the media. For example, a CD allows reading at 50x real time, while a DVD allows reading at 20x real time resulting in the data rates shown. It should be noted that technology changes continuously. Therefore, this table should be taken as a snapshot of technology circa

[2] Slow speed modems of yester years with data rates of 2400 bits/s may have been amenable to PIO with interrupts. However, modern cable modems that support upstream data rates in excess of 1 Mbits/s, and downstream bandwidth in excess of 8 Mbits/s require DMA transfer to avoid data loss.
[3] This assumes an Inkjet print speed of 20 pages per min (ppm) for text and 2-4 ppm for graphics.
[4] Inkjet technology allows pausing printing awaiting data from the computer. Since the data rate is slow enough, it is conceivable to use PIO for this device.
[5] This assumes a Laser print speed of 40 ppm for text and about 4-8 ppm for graphics.
[6] Typically, speakers are in the range of outputting 120 words/minute.
[7] Many thanks to Yale Patt, UT-Austin, and his colleague for shedding light on the speed of peripheral devices.

2008, just as a way of understanding how computer peripherals may be interfaced with the CPU.

10.8 Disk Storage

We will study disk as a concrete example of an important peripheral device. Disk drives are the result of a progression of technology that started with magnetically recording analog data onto wire. This led to recording data on to a magnetic coating applied to Mylar tape. Tapes only allowed sequential access to the recorded data. To increase the data transfer rate as well as to allow random access, magnetic recording transitioned to a rotating drum. The next step in the evolution of magnetic recording was the disk.

Modern disk drives typically consist of some number of *platters* of a lightweight non-ferromagnetic metal coated with a ferromagnetic material on both the top and bottom *surfaces* (see Figure 10.10), thus allowing both surfaces for storing data. A central spindle gangs these platters together and rotates them at very high speed (~15,000 RPM in state-of-the-art high-capacity drives). There is an array of magnetic *read/write heads*, one per surface. The heads do not touch the surfaces. There is a microscopic air gap (measured in nanometers and tinier than a smoke or dust particle) between the head and the surface to allow the movement of the head over the surface without physical contact with the surface. An *arm* connects each head to a common *shaft* as shown in the figure. The shaft, the arms, and the attached heads form the *head assembly*. The arms are mechanically fused together to the shaft to form a single aligned structure such that all the heads can be moved in unison (like a swing door) in and out of the disk. Thus, all the heads simultaneously line up on their respective surfaces at the **same** radial position.

Depending on the granularity of the *actuator* that controls the motion of the head assembly, and the *recording density* allowed by the technology, this arrangement leads to configuring each recording surface into *tracks,* where each track is at a certain pre-defined radial distance from the center of the disk. As the name suggest, a track is a circular band of magnetic recording material on the platter. Further, each track consists of *sectors*. A sector is a contiguous recording of bytes of information, fixed in size, and forms the basic *unit of recording* on the disk. In other words, a sector is the smallest unit of information that can be read or written to the disk. Sensors around the periphery of the disk platters demarcate the sectors. The set of corresponding tracks on all the surfaces form a *logical cylinder* (see Figure 10.11). Cylinder is an aggregation of the corresponding tracks on all the surfaces. The reason for identifying cylinder as a logical entity will become evident shortly when we discuss the latencies involved in accessing the disk and performing I/O operations. The entire disk (platters, head assembly, and sensors) is vacuum-sealed since even the tiniest of dust particles on the surface of the platters will cause the disk to fail.

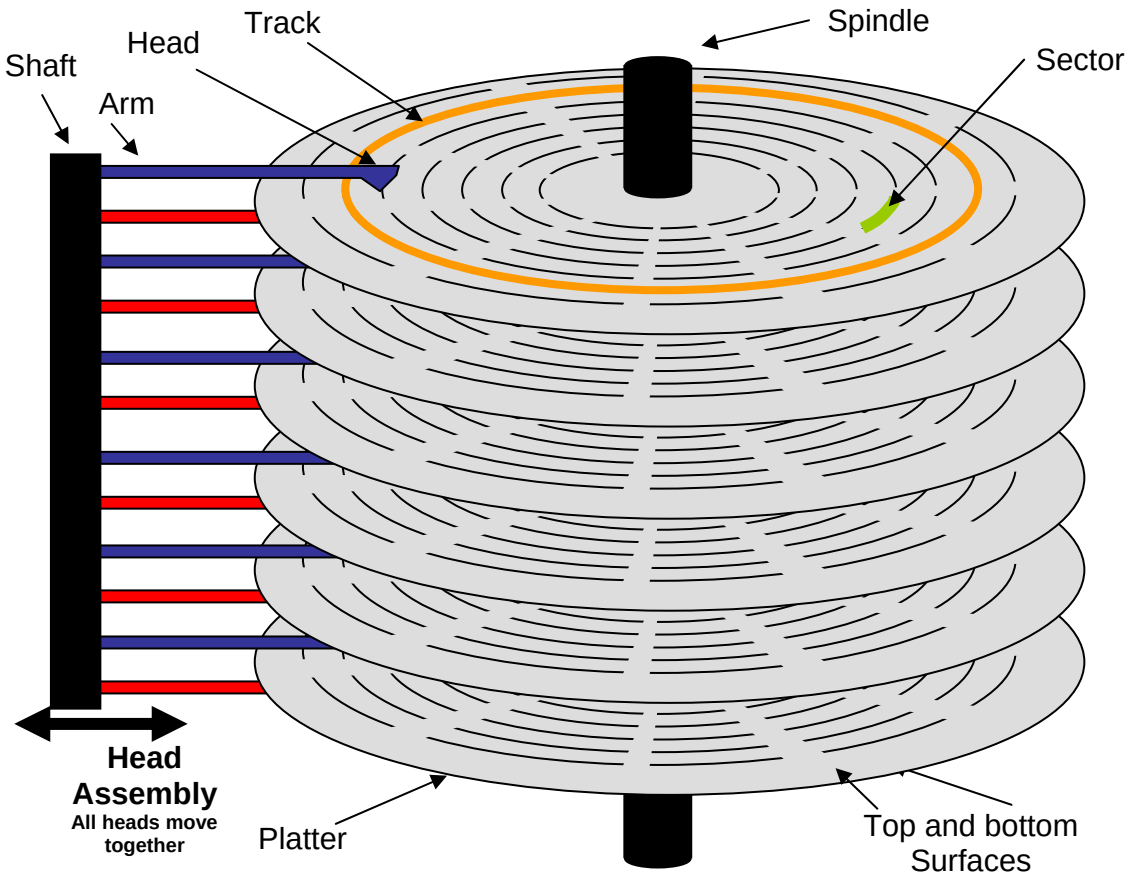


Figure 10.10: Magnetic Disk

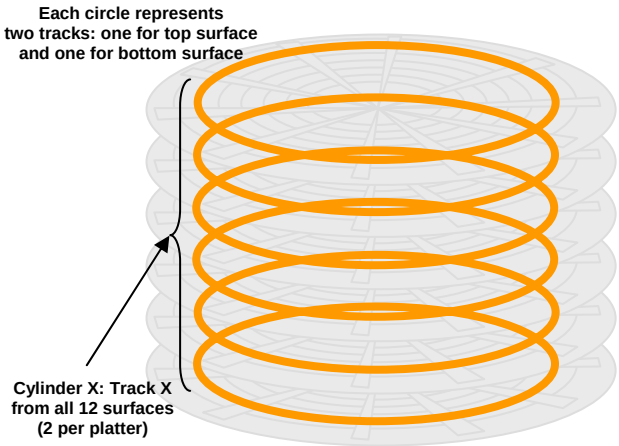
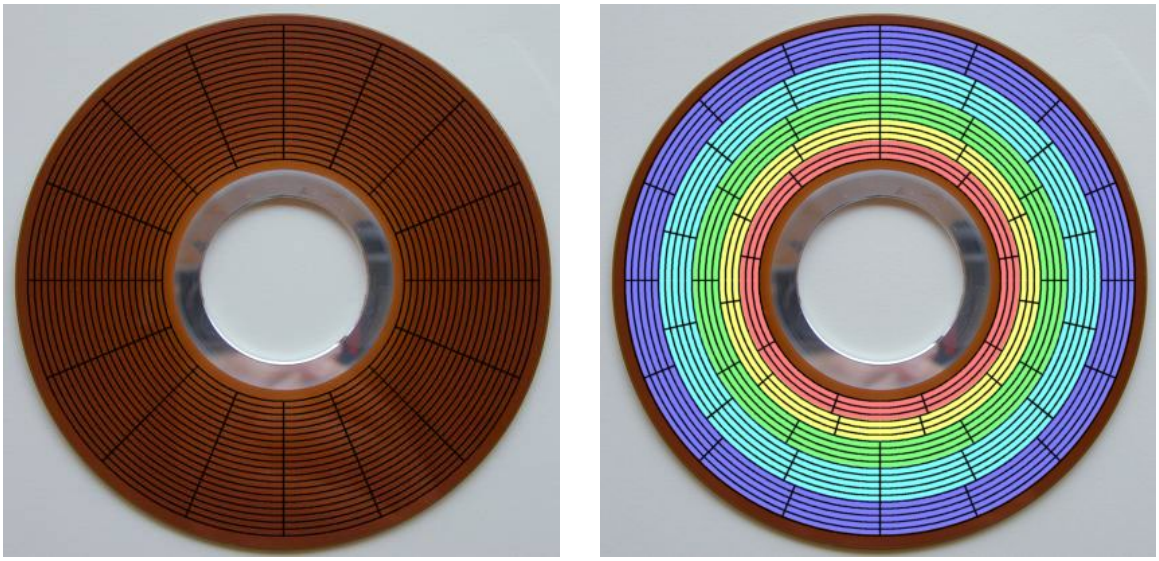


Figure 10.11: A Logical Cylinder for a disk with 6 platters

Tracks are concentric bands on the surface of the disk platters. Naturally, the outer tracks have larger circumference compared to the inner tracks. Therefore, the outer sectors have a larger footprint compared to the inner ones (see Figure 10.12a). As we mentioned, a

sector has a fixed size in terms of recorded information. To reconcile the larger sector footprint and the fixed sector size in the outer tracks, earlier disk technology took the approach of reducing the recording density in the outer tracks. This results in under-utilizing the available space on the disk.



(a) Normal (Non zoned) recording **(b) Zoned recording**

Figure 10.12: Difference between non-zoned and zoned recording[8]

To overcome this problem of under-utilization, modern disk drives use a technology called *zoned bit recording (ZBR)* that keeps the footprint of the sectors roughly constant on the entire surface. However, the outer tracks have more sectors than the inner ones (see Figure 10.12b). The disk surface is divided into zones; tracks in the different zones have different number of sectors thus resulting in a better utilization.

Let,

p – number of platters,
n – number of surfaces per platter (1 or 2),
t – number of tracks per surface,
s – number of sectors per track,
b – number of bytes per sector,

The total capacity of the disk assuming non-zoned recording:

$$\text{Capacity} = (p * n * t * s * b) \text{ bytes} \quad (1)$$

[8] Picture source: http://www.pcguide.com/ref/hdd/geom/tracksZBR-c.html

With zoned recording,

z – number of zones,
t_{zi} – number of tracks at zone z_i,
s_{zi} – number of sectors per track at zone z_i,

The total capacity of the disk with zoned recording:

$$\text{Capacity} = (p * n * (\sum (t_{zi} * s_{zi}), \text{ for } 1 \leq i \leq z) * b) \text{ bytes} \quad (2)$$

Example 1:

Given the following specifications for a disk drive:

- 256 bytes per sector
- 12 sectors per track
- 20 tracks per surface
- 3 platters

a) What is the total capacity of such a drive in bytes assuming normal recording?

b) Assume a zoned bit recording with the following specifications:
   - 3 zones
     - Zone 3 (outermost): 8 tracks, 18 sectors per track
     - Zone 2: 7 tracks, 14 sectors per track
     - Zone 1: 5 tracks, 12 sectors per track

   What is the total capacity of this drive with the zoned-bit recording?

Answer:

a)

Total capacity
= number of platters * surfaces/platter * tracks/surface * sectors/track * bytes/sector
= 3 * 2 * 20 * 12 * 256 bytes
= **360 Kbytes (where K = 1024)**

b)

Capacity of Zone 3 =
= number of platters * surfaces/platter * tracks in zone 3 * sectors/track * bytes/sector
= 3 * 2 * 8 * 18 * 256
= 216 Kbytes
Capacity of Zone 2 =
= number of platters * surfaces/platter * tracks in zone 2 * sectors/track * bytes/sector
= 3 * 2 * 7 * 14 * 256
= 147 Kbytes
Capacity of Zone 1 =
= number of platters * surfaces/platter * tracks in zone 1 * sectors/track * bytes/sector
= 3 * 2 * 5 * 12 * 256
= 90 Kbytes

Total capacity = sum of all zonal capacities = 216 + 147 + 90
= **453 Kbytes (where K = 1024)**

An important side effect of ZBR is the difference in transfer rates between outer and inner tracks. The outer track has more sectors compared to the inner ones; and the angular velocity of the disk is the same independent of the track being read. Therefore, the head reads more sectors per revolution of the disk when it is over an outer track compared to an inner track. In allocating space on the disk, there is a tendency to use the outer tracks first before using the inner tracks.

The address of a particular data block on the disk is a triple: {*cylinder#, surface#, sector#*}. Reading or writing information to or from the disk requires several steps. First, the head assembly has to move to the specific cylinder. The time to accomplish this movement is the *seek time*. We can see that seeking to a particular cylinder is the same as seeking to any specific track in that cylinder since a cylinder is just a logical aggregation of the corresponding tracks on all the surfaces (see Figure 10.11). Second, the disk must spin to bring the required sector under the head. This time is the *rotational latency*. Third, data from the selected surface is read and transferred to the controller as the sector moves under the head. This is the *data transfer time*.

Of the three components to the total time for reading or writing to a disk, the seek time is the most expensive, followed by the rotational latency, and lastly the data transfer time. Typical values for seek time and average rotational latency are 8 ms and 4 ms, respectively. These times are so high due to the electro-mechanical nature of the disk subsystem.

Let us understand how to compute the data transfer time. Note that the disk does not stop for reading or writing. Just as in a VCR the tape is continuously moving while the head is reading and displaying the video on the TV, the disk is continually spinning and the head is reading (or writing) the bits off the surface as they pass under the head. The data transfer time is derivable from the rotational latency and the recording density of the media. You are perhaps wondering if reading or writing the media itself does not cost anything. The answer is it does; however, this time is purely electro-magnetic and is negligible in comparison to the electro-mechanical delay in the disk spinning to enable all the bits of a given sector to be read.

We refer to the *data transfer rate* as the amount of data transferred per unit time once the desired sector is under the magnetic reading head. Circa 2008, data transfer rates are in the range of 200-300 Mbytes/sec.

Let,

r – rotational speed of the disk in Revolutions Per Minute (RPM),
s – number of sectors per track,
b – number of bytes per sector,

Time for one revolution = 60/r seconds

Amount of data read in one revolution = $s * b$ bytes

The data transfer rate of the disk:

(Amount of data in track) / (time for one revolution) = $(s * b)/(60/r)$

$$\text{Data transfer rate} = (s * b * r) / 60 \text{ bytes/second} \quad (3)$$

Example 2:

Given the following specifications for a disk drive:

- 512 bytes per sector
- 400 sectors per track
- 6000 tracks per surface
- 3 platters
- Rotational speed 15000 RPM
- Normal recording

What is the transfer rate of the disk?

Answer:

Time for one rotation = 1/15000 minutes
= 4 ms

The amount of data in track = sectors per track * bytes per sector
= 400 * 512
= 204,800 bytes

Since the head reads one track in one revolution of the disk, the transfer rate
= data in one track/time per revolution
= (204,800/4) * 1000 bytes/sec
= **51,200,000 bytes/sec**

The seek time and rotational latency experienced by a specific request depends on the exact location of the data on the disk. Once the head is positioned on the desired sector the time to read/write the data is deterministic, governed by the rotational speed of the disk. It is often convenient to think of *average seek time* and *average rotational latency* in performance estimation of disk drives to satisfy requests. Assuming a uniform distribution of requests over all the tracks, the average seek time is the mean of the observed times to seek to the first track and the last track of the disk. Similarly, assuming a uniform distribution of requests over all the sectors in a given track, the average rotational latency is the mean of the access times to each sector in the track. This is half the rotational latency of the disk[9].

[9] In the best case the desired sector is already under the head when the desired track is reached; in the worst case the head just missed the desired sector and waits for an entire revolution.

Let,

a – average seek time in seconds
r – rotational speed of the disk in Revolutions Per Minute (RPM),
s – number of sectors per track,

Rotational latency = 60/*r* seconds (4)

Average rotational latency = (60 / (*r* * 2)) seconds (5)

Once the read head is over the desired sector, then the time to read that sector is entirely decided by the RPM of the disk.

Sector read time = rotational latency / number of sectors per track

Sector read time = (60 / (*r* * s)) seconds (6)

To read a random sector on the disk, the head has to seek to that particular sector, and then the head has to wait for the desired sector to appear under it, and then read the sector. Thus, there are three components to the time to read a random sector:

- Time to get to the desired track
  = Average seek time
  = *a* seconds
- Time to get the head over the desired sector
  = Average rotational latency
  = (60 / (*r* * 2)) seconds
- Time to read a sector
  = Sector read time
  = (60 / (*r* * *s*)) seconds

Time to read a random sector on the disk (7)
= Time to get to the desired track +
Time to get the head over the desired sector +
Time to read a sector
= *a* + (60 / (*r* * 2)) + (60 / (*r* * *s*)) seconds

Example 3:

Given the following specifications for a disk drive:

- 256 bytes per sector
- 12 sectors per track
- 20 tracks per surface
- 3 platters
- Average seek time of 20 ms
- Rotational speed 3600 RPM
- Normal recording

a) What would be the time to read 6 contiguous sectors from the same track?
b) What would be the time to read 6 sectors at random?

Answer:

a)
Average seek time = 20 ms
Rotational latency of the disk
= 1/3600 minutes
= 16.66 ms
Average rotational latency
= rotational latency/2
= 16.66/2
Time to read 1 sector
= rotational latency / number of sectors per track
= 16.66/12 ms

To read 6 contiguous sectors on the same track the time taken
= 6 * (16.66/12)
= 16.66/2

Time to read 6 contiguous sectors from the disk
= average seek time + average rotational latency + time to read 6 sectors
= 20 ms + 16.66/2 + 16.66/2
= **36.66 ms**

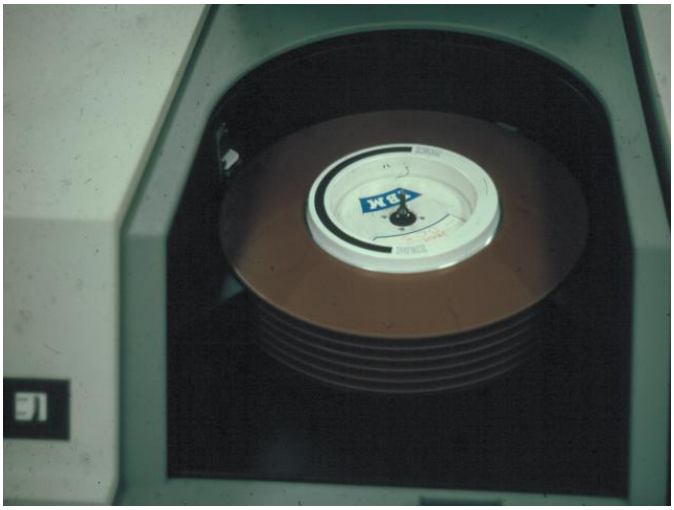
b)
For the second case, we will have to seek and read each sector separately.

Therefore, each sector read will take
= average seek time + average rotational latency + time to read 1 sector
= 20 + 16.66/2 + 16.66/12

Thus total time to read 6 random sectors
= 6 * (20 + 16.66/2 + 16.66/12)
= **178.31 ms**

10.8.1 Saga of Disk Technology

The discussion until now has been kept simple just for the sake of understanding the basic terminologies in disk subsystems. The disk technology has seen an exponential growth in recording densities for the past two decades. For example, circa 1980, 20 megabytes was considered a significant disk storage capacity. Such disks had about 10 platters and were bulky. The drive itself looked like a clothes washer (see Figure 10.13) and the media was usually removable from the drive itself.



(a) Removable Media (b) Disk drive

Figure 10.13: Magnetic media and Disk drive[10]

Circa 2008, a desktop PC comes with several 100 Gigabytes of storage capacity. Such drives have the media integrated into them. Circa 2008, small disks for the desktop PC market (capacity roughly 100 GB to 1 TB, 3.5" diameter) have 2 to 4 platters, rotation speed in the range of 7200 RPM, 5000 to 10000 tracks per surface, several 100 sectors per track, and 256 to 512 bytes per sector (see Figure 10.14).

Figure 10.14: PC hard drive (circa 2008)[11]

[10] Picture source: unknown

[11] Picture of a Western Digital Hard Drive: variable RPM, 1 TB capacity, source: http://www.wdc.com

While we have presented fairly simple ways to compute the transfer rates of disks and a model of accessing the disk based on the cylinder-track-sector concept, modern technology has thrown all these models out of whack. The reason for this is fairly simple. Internally the disk remains the same as pictured in Figure 10.14. Sure, the recording densities have increased, the RPM has increased, and consequently the size of the platter as well as the number or platters in a drive has dramatically decreased. These changes themselves do not change the basic model of disk access and accounting for transfer times. The real changes are in three areas: advances in the drive electronics, advances in the recording technology, and advance in interfaces.

The first advance is in the internal drive electronics. The simple model assumes that the disk rotates at a constant speed. The head moves to the desired track and waits for the desired sector to appear under it. This is wasteful of power. Therefore, modern drives vary the RPM depending on the sector that needs to be read, ensuring that the sector appears under the head just as the head assembly reaches the desired track.

Another advance is in the recording strategy. Figure 10.15 shows a cross section of the magnetic surface to illustrate this advance in recording technology. Traditionally, the medium records the data bits by magnetizing the medium horizontally parallel to the magnetic surface sometimes referred to as *longitudinal* recording (Figure 10.15-(a)). A recent innovation in the recording technology is the *perpendicular magnetic recording (PMR)*, which as the name suggest records the data bits by magnetizing the medium perpendicular to the magnetic surface (Figure 10.15-(b)). Delving into the electromagnetic properties of these two recording techniques is beyond the scope of this book. The intent is to get across the point that this new technology greatly enhances the recording density achievable per unit area of the magnetic surface on the disk. Figure 10.15 illustrates this point. It can be seen that PMR results in doubling the recording density and hence achieves larger storage capacity than longitudinal recording for a given disk specification.

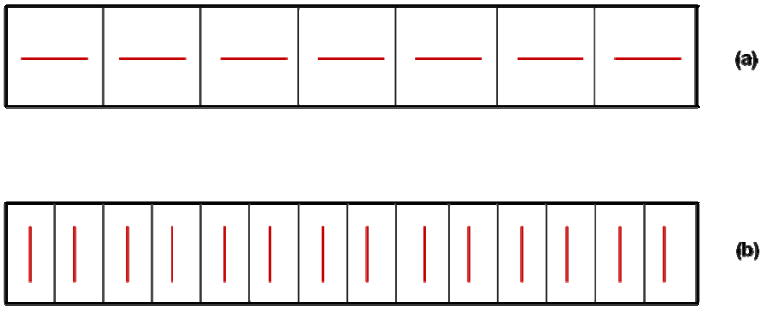


Figure 10.15: Disk recording: (a) Longitudinal recording; (b) PMR

The third change is in the computational capacity that has gotten into these modern drives. Today, a hard drive provides much more intelligent interfaces for connecting it to the rest of the system. Gone are the days when a disk controller sits outside the drive. Today the drive is integrated with the controller. You may have heard of terminologies such as IDE (which stands for *Integrated Drive Electronics*), ATA (which stands for *Advanced Technology Attachment*), SATA (which stands for *Serial Advanced*

Technology Attachment) and SCSI (which stands for *Small Computer Systems Interface*). These are some of the names for modern intelligent interfaces.

Conceptually, these advanced interfaces reduce the amount of work the CPU has to do in ordering its requests for data from the disk. Internal to the drive is a microprocessor that decides how logically contiguous blocks (which would be physically contiguous as well in the old world order) may be physically laid out to provide best access time to the data on the disk. In addition to the microprocessors, the drives include data buffers into which disk sectors may be pre-read in readiness for serving requests from the CPU. The microprocessor maintains an internal queue of requests from the CPU and may decide to reorder the requests to maximize performance.

Much of the latency in a disk drive arises from the mechanical motions involved in reading and writing. Discussion of the electromechanical aspects of disk technology is beyond the scope of this book. Our focus is on how the system software uses the disk drive for storing information. The storage allocation schemes should try to reduce the seek time and the rotational latency for accessing information. Since we know that seek time is the most expensive component of disk transfer latency, we can now elaborate on the concept of a logical cylinder. If we need to store a large file that may span several tracks, should we allocate the file to **adjacent tracks on the same surface** or **corresponding (same) tracks of a given cylinder**? The answer to this question has become quite complex with modern disk technology. As a first order, we will observe that the former will result in multiple seeks to access different parts of the given file while the latter will result in a single seek to access any part of a given file. We can easily see that the latter allocation will be preferred. This is the reason for recognizing a logical cylinder in the context of the disk subsystem. Having said that, we should observe that since the number of platters in a disk is reducing and is typically 1 or 2, the importance of the cylinder concept is diminishing.

File systems is the topic of the next chapter, wherein we elaborate on storage allocation schemes.

From the system throughput point of view, the operating system should schedule operations on the disk in such a manner as to reduce the overall mechanical motion of the disk. In modern drives, this kind of reordering of requests happens within the drive itself. Disk scheduling is the topic of discussion in the next section.

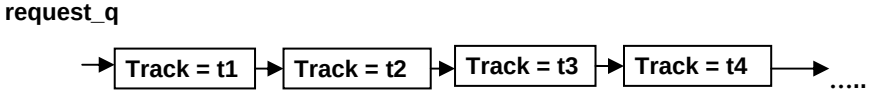


Figure 10.16: Disk request queue in the order of arrival

10.9 Disk Scheduling Algorithms

A device driver for a disk embodies algorithms for efficiently scheduling the disk to satisfy the requests that it receives from the operating system. As we have seen in earlier

chapters (see Chapters 7 and 8), the memory manager component of the operating system may make its own requests for disk I/O to handle demand paging. As we will see shortly in Chapter 11, disk drives host file systems for end users. Thus, the operating system (via system calls) may issue disk I/O requests in response to users' request for opening, closing, and reading/writing files. Thus, at any point of time, the device driver for a disk may be entertaining a number of I/O requests from the operating system (see Figure 10.16). The operating system queues these requests in the order of generation. The device driver schedules these requests commensurate with the disk-scheduling algorithm in use. Each request will name among other things, the specific track where the data resides on the disk. Since seek time is the most expensive component of I/O to or from the disk, the primary objective of disk scheduling is to minimize seek time.

We will assume for this discussion that we are dealing with a single disk that has received a number of requests and must determine the most efficient algorithm to process those requests. We will further assume that there is a single head and that seek time is proportional to the number of tracks crossed. Finally, we will assume a random distribution of data on the disk, and that reads and writes take equal amounts of time.

The typical measures of how well the different algorithms stack up against one another are the *average waiting time* for a request, *the variance in wait time* and the overall *throughput*. Average wait time and throughput are self-evident terms from our discussion on CPU scheduling (Chapter 6). These terms are system centric measures of performance. From an individual request point of view, variance in waiting time is more meaningful. This measure tells us how much an individual request's waiting time can deviate from the average. Similar to CPU scheduling, the *response time* or *turnaround time* is a useful metric from the point of view of an individual request.

In Table 10.3, t_i, w_i, and e_i, are the turnaround time, wait time, and actual I/O service time, for an I/O request i, respectively. Most of these metrics and their mathematical notation are similar to the ones given in Chapter 6 for CPU scheduling.

Name	Notation	Units	Description
Throughput	n/T	Jobs/sec	System-centric metric quantifying the number of I/O requests *n* executed in time interval *T*
Avg. Turnaround time (t_{avg})	$(t_1+t_2+\ldots+t_n)/n$	Seconds	System-centric metric quantifying the average time it takes for a job to complete
Avg. Waiting time (w_{avg})	$((t_1-e_1) + (t_2-e_2)+ \ldots + (t_n-e_n))/n$ or $(w_1+w_2+ \ldots +w_n)/n$	Seconds	System-centric metric quantifying the average waiting time that an I/O request experiences
Response time/ turnaround time	t_i	Seconds	User-centric metric quantifying the turnaround time for a specific I/O request *i*
Variance in Response time	$E[(t_i - t_{avg})^2]$	$Seconds^2$	User-centric metric that quantifies the statistical variance of the actual response time (t_i) experienced by an I/O request *i* from the expected value (t_{avg})
Variance in Wait time	$E[(w_i - w_{avg})^2]$	$Seconds^2$	User-centric metric that quantifies the statistical variance of the actual wait time (w_i) experienced by an I/O request *i* from the expected value (w_{avg})
Starvation	-	-	User-centric qualitative metric that signifies denial of service to a particular I/O request or a set of I/O request due to some intrinsic property of the I/O scheduler

Table 10.3: Summary of Performance Metrics

We review five different algorithms for disk scheduling. To make the discussion simple and concrete, we will assume that the disk has 200 tracks numbered 0 to 199 (with 0 being the outermost and 199 being the innermost). The head in its fully retracted position

is on track 0. The head assembly extends to its maximum span from its resting position when it is on track 199.

You will see a similarity between these algorithms and some of the CPU scheduling algorithms we saw in Chapter 6.

10.9.1 First-Come First Served

As the name suggests, this services the requests by the order of arrival. In this sense, it is similar to the FCFS algorithm for CPU scheduling. The algorithm has the nice property that a request incurs the least variance in waiting time regardless of the track from which it requests I/O. However, that is the only good news. From the system's perspective, this algorithm will result in poor throughput for most common workloads. Figure 10.17 illustrates how the disk head may have to swing back and forth across the disk surface to satisfy the requests in an FCFS schedule, especially when the FCFS requests are to tracks that are far apart.

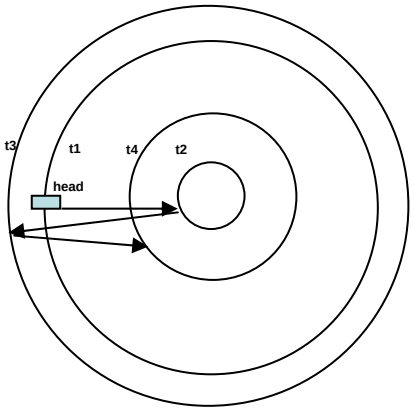


Figure 10.17: Movement of Disk Head with FCFS

10.9.2 Shortest Seek Time First

This scheduling policy has similarity to the SJF processor scheduling. The basic idea is to service the tracks that lead to minimizing the head movement (see Figure 10.18). As with SJF for processor scheduling, SSTF minimizes the average wait time for a given set of requests and results in good throughput. However, similar to SJF it has the potential of starving requests that happen to be far away from the current cluster of requests. Compared to FCFS, this schedule has high variance.

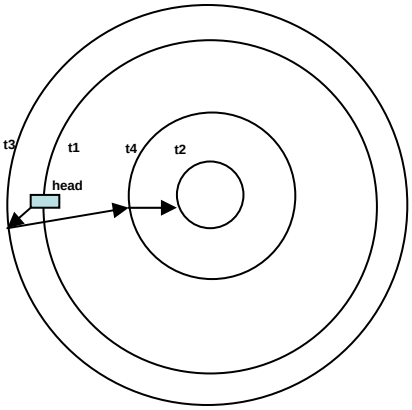


Figure 10.18: Movement of Disk Head with SSTF

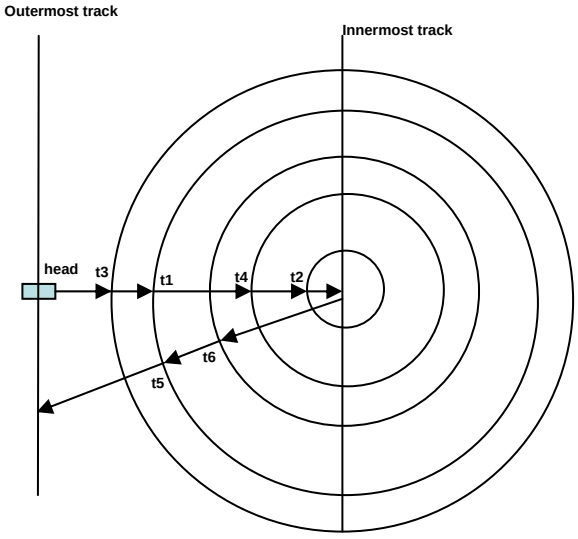


Figure 10.19: Movement of Disk Head with SCAN

10.9.3 Scan (elevator algorithm)

This algorithm is in tune with the electro-mechanical properties of the head assembly. The basic idea is as follows: the head moves from its position of rest (track 0) towards the innermost track (track 199). As the head moves, the algorithm services the requests that are *en route* from outermost to innermost track, regardless of the arrival order. Once the head reaches the innermost track, it reverses direction and moves towards the outermost track, once again servicing requests en route. As long as the request queue is non-empty, the algorithm continuously repeats this process. Figure 10.19 captures the spirit of this algorithm.

The requests t1, t2, t3, and t4 existed during the forward movement of the head (as in Figure 10.16). Requests t5 and t6 (in that order) appeared after the head reached the innermost track. The algorithm services these requests on the reverse traversal of the

head. This algorithm should remind you of what happens when you wait for an elevator, which uses a similar algorithm for servicing requests. Hence, the SCAN algorithm is often referred to as the elevator algorithm. SCAN has lower variance in wait time compared to SSTF, and overall has an average wait time that is similar to SSTF. Similar to SSTF, SCAN does not preserve the arrival order of the requests. However, SCAN differs from SSTF in one fundamental way. SSTF may starve a given request arbitrarily. On the other hand, there is an upper bound for SCAN's violation of the first-come-first-served fairness property. The upper bound is the traversal time of the head from one end to the other. Hence, SCAN avoids starvation of requests.

10.9.4 C-Scan (Circular Scan)

This is a variant of SCAN. The algorithm views the disk surface as logically circular. Hence, once the head reaches the innermost track, the algorithm retracts the head assembly to the position of rest, and the traversal starts all over again. In other words, the algorithm does not service any requests during the traversal of the head in the reverse direction. This pictorially shown for the same requests serviced in the SCAN algorithm in Figure 10.20.

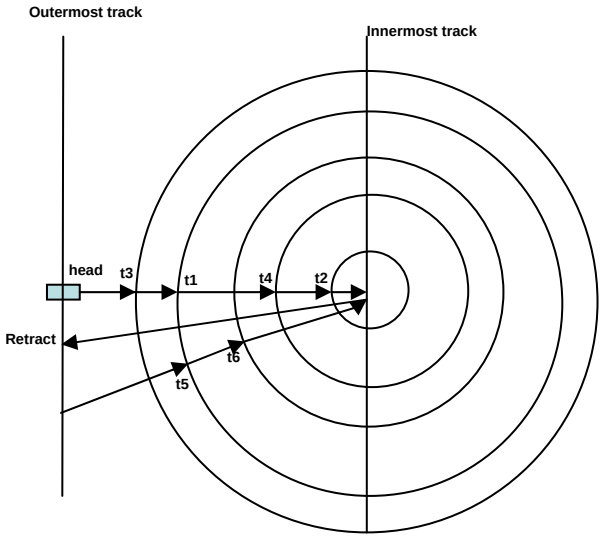
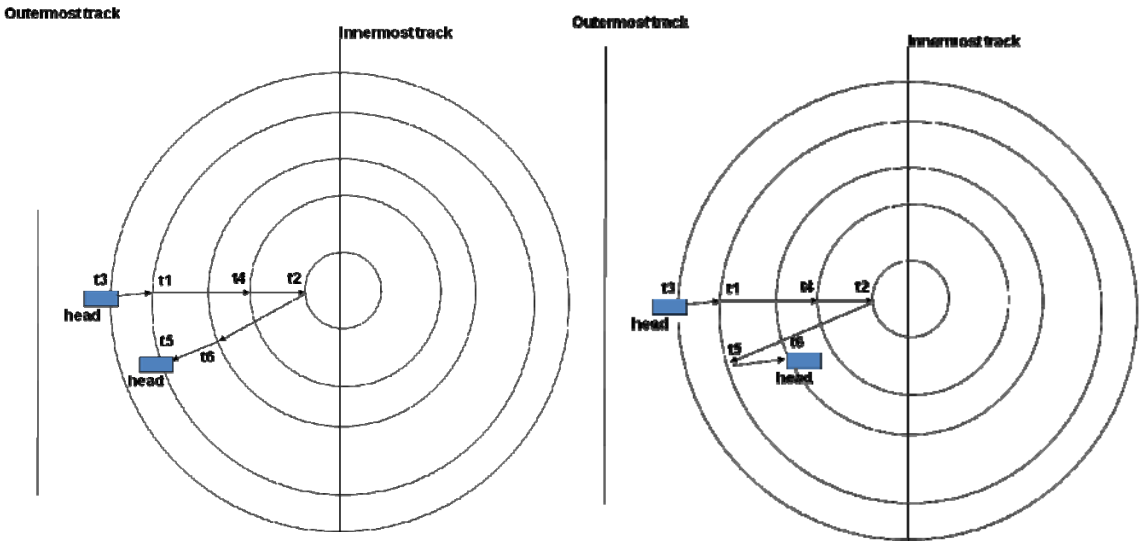


Figure 10.20: Movement of Disk Head with C-SCAN

By ignoring requests in the reverse direction, C-SCAN removes the bias that the SCAN algorithm has for requests clustered around the middle tracks of the disk. This algorithm reduces unfairness in servicing requests (notice how it preserves the order of arrival for t5 and t6), and overall leads to lowering the variance in waiting time compared to SCAN.

10.9.5 Look and C-Look

These two policies are similar to Scan and C-Scan, respectively; with the exception that if there are no more requests in the direction of head traversal, then the head assembly immediately reverses direction. That is, the head assembly does not unnecessarily traverse all the way to the end (in either direction). One can see the similarity to how an elevator works. Due to avoiding unnecessary mechanical movement, these policies tend to be even better performers than the Scan and C-Scan policies. Even though, historically, SCAN is referred to as the elevator algorithm, LOOK is closer to the servicing pattern observed in most modern day elevator systems. Figure 10.21 shows the same sequence of requests as in the SCAN and C-SCAN algorithms for LOOK and C-LOOK algorithms. Note the position of the head at the end of servicing the outstanding requests. The head stops at the last serviced request if there are no further requests. This is the main difference between LOOK and SCAN, and C-LOOK and C-SCAN, respectively.



(a) LOOK **(b) C-LOOK**

Figure 10.21: Movement of Disk Head with LOOK and C-LOOK

10.9.6 Disk Scheduling Summary

The choice of the scheduling algorithm depends on a number of factors including expected layout, the storage allocation policy, and the electromechanical capabilities of the disk drive. Typically, some variant of LOOK or C-LOOK is used for disk scheduling. We covered the other algorithms more from the point of completeness than as viable choices for implementation in a real system.

As we mentioned in Section 10.8.1, modern disk drives provide very sophisticated interface to the CPU. Thus, the internal layout of the blocks on the drive may not even be visible to the disk device driver, which is part of the operating system. Assuming that the

interface allows multiple outstanding requests from the device driver, the disk scheduling algorithms discussed in this section are in the controller itself.

Example 4 illustrates the difference in schedule of the various algorithms.

Example 4:
Given the following:

Total number of cylinders in the disk	=	200 (numbered 0 to 199)
Current head position	=	cylinder 23
Current requests in order of arrival	=	20, 17, 55, 35, 25, 78, 99

Show the schedule for the various disk scheduling algorithms for the above set of requests.

Answer:

(a) Show the schedule for C-LOOK for the above requests

25, 35, 55, 78, 99, 17, 20

(b) Show the schedule for SSTF

25, 20, 17, 35, 55, 78, 99

(c) Show the schedule for LOOK

25, 35, 55, 78, 99, 20, 17

(d) Show the schedule for SCAN

25, 35, 55, 78, 99, 199, 20, 17, 0

(e) Show the schedule for FCFS

20, 17, 55, 35, 25, 78, 99

(f) Show the schedule for C-SCAN

25, 35, 55, 78, 99, 199, 0, 17, 20

10.9.7 Comparison of the Algorithms

Let us analyze the different algorithms using the request pattern in Example 4. In the order of arrival, we have seven requests: R1 (cylinder 20), R2 (cylinder 17), R3 (cylinder 55), R4 (cylinder 35), R5 (cylinder 25), R6 (cylinder 78), and R7 (cylinder 99).

Let us focus on request R1. We will use the number of tracks traversed as the unit of response time for this comparative analysis. Since the starting position of the head in this example is 23, the response time for R1 for the different algorithms:

- $T_1^{FCFS} = 3$ (R1 gets serviced first)
- $T_1^{SSTF} = 7$ (service R5 first and then R1)
- $T_1^{SCAN} = 355$ (sum up the head traversal for (d) in the above example)
- $T_1^{C\text{-}SCAN} = 395$ (sum up the head traversal for (f) in the above example)
- $T_1^{LOOK} = 155$ (sum up the head traversal for (c) in the above example)
- $T_1^{C\text{-}LOOK} = 161$ (sum up the head traversal for (a) in the above example)

Table 10.4 shows a comparison of the response times (in units of head traversal) with respect to the request pattern in Example 4 for some chosen disk scheduling algorithms (FCFS, SSTF, and LOOK).

Requests	Response time		
	FCFS	SSFT	LOOK
R1 (cyl 20)	3	7	155
R2 (cyl 17)	6	10	158
R3 (cyl 55)	44	48	32
R4 (cyl 35)	64	28	12
R5 (cyl 25)	74	2	2
R6 (cyl 78)	127	71	55
R7 (cyl 99)	148	92	76
Average	66.4	36	70

Table 10.4: Quantitative Comparison of Scheduling Algorithms for Example 4

The throughput is computed as the ratio of the number of requests completed to the total number of tracks traversed by the algorithm to complete all the requests.

- FCFS = 7/148 = 0.047 requests/track-traversal
- SSTF = 7/92 = 0.076 requests/track-traversal
- LOOK = 7/158 = 0.044 requests/track-traversal

From the above analysis, it would appear that SSTF does the best in terms of average response time and throughput. But this comes at the cost of fairness (compare the response times for R5 with respect to earlier request R1-R4 in the column under SSTF). Further, as we observed earlier SSTF has the potential for starvation. At first glance, it would appear that LOOK has the worst response time of the three compared in the table. However, there a few points to note. First, the response times are sensitive to the initial head position and the distribution of requests. Second, it is possible to come up with a pathological example that may be particularly well suited (or by the same token ill-suited) for a particular algorithm. Third, in this contrived example we started with a request sequence that did not change during the processing of this sequence. In reality,

new requests may join the queue that would have an impact on the observed throughput as well as the response time (see Exercise 11).

In general, with uniform distribution of requests on the disk the average response time of LOOK will be closer to that of SSTF. More importantly, something that is not apparent from the table is both the time and energy needed to change the direction of motion of the head assembly that is inherent in both FCFS and SSTF. This is perhaps the single most important consideration that makes LOOK a more favorable choice for disk scheduling.

Using Example 4, it would be a useful exercise for the reader to compare all the disk scheduling algorithms with respect to the other performance metrics that are summarized in Table 10.3.
Over the years, disk-scheduling algorithms have been studied extensively. As we observed earlier (Section 10.8.1), disk drive technology has been advancing rapidly. Due to such advances, it becomes imperative to re-evaluate the disk-scheduling algorithms with every new generation of disks[12]. So far, some variant of LOOK has proven to perform the best among all the candidates.

10.10 Solid State Drive

One of the fundamental limitations of the hard disk technology is its electromechanical nature. Over the years, several new storage technologies have posed a threat to the disk but they have not panned out primarily due to the low cost per byte of disk compared to these other technologies.

Figure 10.22: A Railroad Switch[13]

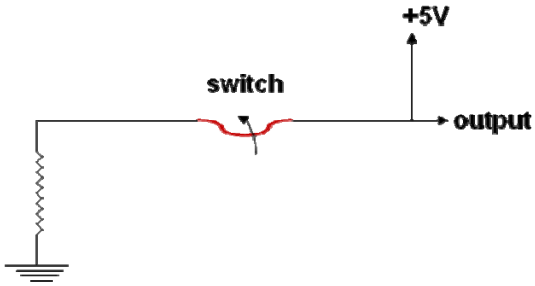


Figure 10.23: An Electrical Switch

A technology that is threatening the relative monopoly of the hard disk is *Solid State Drive (SSD)*. Origins of this technology can be traced back to *Electrically Erasable Programmable Read-Only Memory (EEPROM)*. In Chapter 3, we introduced ROM as a kind of solid state memory whose contents are *non-volatile*, i.e., the contents remain unchanged across power cycling. It will be easy to understand this technology with a simple analogy. You have seen a railroad switch shown in Figure 10.22. Once the

[12] See for example, http://www.ece.cmu.edu/~ganger/papers/sigmetrics94.pdf
[13] Source: http://upload.wikimedia.org/wikipedia/commons/d/da/Railway_turnout_-_Oulu_Finland.jpg

switch is thrown the incoming track (bottom part of the figure) remains connected to the chosen fork. A ROM works in exactly the same manner.

Figure 10.23 is a simple electrical analogy of the railroad switch. If the switch in the middle of the figure is opened then the output is a 1; otherwise the output is a 0. This is the basic building block of a ROM. The switches are realized using basic logic gates (NAND or NOR). Collection of such switches packed inside an integrated circuit is the ROM. Depending on the desired output, a switch can be "programmed" so that the output is a 0 or a 1. This is the reason such a circuit is referred to as *Programmable Read-Only Memory (PROM)*.

A further evolution in this technology allows the bit patterns in the ROM to be electrically erased and re-programmed to contain a different bit pattern. This is the EEPROM technology. While this technology appears very similar to the capability found in a RAM, there is an important difference. Essentially, the granularity of read/write in a RAM can be whatever we choose it to be. However, due to the electrical properties of the EEPROM technology, erasure in an EEPROM needs to happen a block of bits at a time. Further, such writes take several orders of magnitude more time compared to reading and writing a RAM. Thus, EEPROM technology cannot be used in place of the DRAM technology. However, this technology popularized as *Flash memory* found a place in portable memory cards, and for storage inside embedded devices such as cell phones and iPods.

A logical question that may perplex you is why Flash memory has not replaced the disk as permanent storage in desktops and laptops. After all, being entirely solid state, this technology does not have the inherent problems of the disk technology (slower access time for data due to the electromechanical nature of the disk).

Three areas where the hard disk still has an edge are density of storage (leading to lower cost per byte), higher read/write bandwidth, and longer life. The last point needs some elaboration. SSD, due to the nature of the technology, has an inherent limitation that any given area of the storage can be rewritten only a finite number of times. This means that frequent writes to the same block would lead to uneven wear, thus shortening the lifespan of the storage as a whole. Typically, manufacturers of SSD adopt *wear leveling* to avoid this problem. The idea with wear leveling is to redistribute frequently written blocks to different sections of the storage. This necessitates additional read/write cycles over and beyond the normal workload of the storage system.

Technological advances in SSD are continually challenging these gaps. For example, circa 2008, SSD devices with a capacity of 100 GB and transfer rates of 100 MB/second are hitting the market place. Several box makers are introducing laptops with SSD as the mass storage for low-end storage needs. Circa 2008, the cost of SSD-based mass storage is significantly higher than a comparable disk-based mass storage.

Only time will tell whether the hard drive technology will continue its unique long run as the ultimate answer for massive data storage.

10.11 Evolution of I/O Buses and Device Drivers

With the advent of the PC, connecting peripherals to the computer has taken a life of its own. While the data rates in the above table suggest how the devices could be interfaced to the CPU, it is seldom the case that devices are directly interfaced to the CPU. This is because peripherals are made by "third party vendors[14]," a term that is used in the computer industry to distinguish "box makers" such as IBM, Dell, and Apple from the vendors (e.g., Seagate for disks, Axis for cameras, etc.) who make peripheral devices and device drivers that go with such devices. Consequently, such third party vendors are not privy to the internal details of the boxes and therefore have to assume that their wares could be interfaced to any boxes independent of the manufacturer. You would have heard the term "plug and play". This refers to a feature that allows any peripheral device to be connected to a computer system without doing anything to the internals of the box. This feature has been the primary impetus for the development of standards such as PCI, which we introduced in Section 10.4.

In recent times, it is impossible for a computer enthusiast not to have heard of terminologies such as *USB* and *Firewire*. Let us get familiar with these terminologies. As we mentioned in Section 10.4, PCI bus uses 32-bit parallel bus that it multiplexes for address, data, and command. USB which stands for *Universal Serial Bus*, and *Firewire* are two competing standards for *serial* interface of peripherals to the computer. You may wonder why you would want a serial interface to the computer, since a parallel interface would be faster. In fact, if one goes back in time, only slow speed character-oriented devices (such as Cathode Ray Terminal or CRT, usually referred to as "dumb terminals") used a serial connection to the computer system.

Well, ultimately, you know that the speed of the signals on a single wire is limited by the *speed of light*. The actual data rates as you can see from Table 10.2 are nowhere near that. Latency of electronic circuitry has been the limiting factor in pumping the bits faster on a single wire. Parallelism helps in overcoming this limitation and boosts the overall throughput by pumping the bits on parallel wires. However, as technology improves the latency of the circuitry has been reducing as well allowing signaling to occur at higher frequencies. Under such circumstances, serial interfaces offer several advantages over parallel. First, it is smaller and cheaper due to the reduced number of wires and connectors. Second, parallel interfaces suffer from cross talk at higher speeds without careful shielding of the parallel wires. On the other hand, with careful wave-shaping and filtering techniques, it is easier to operate serial signaling at higher frequencies. This has led to the situation now where serial interfaces are actually faster than parallel ones.

Thus, it is becoming the norm to connect high-speed devices serially to the computer system. This was the reason for the development of standards such as USB and Firewire for serially interfacing the peripherals to the computer system. Consequently, one may notice that most modern laptops do not support any parallel ports. Even a parallel printer

[14] We already introduced this term without properly defining it in Section 10.4.

port has disappeared from laptops post 2007. Serial interface standards have enhanced the "plug and play" nature of modern peripheral devices.

You may wonder why there are two competing standards. This is again indicative of the "box makers" wanting to capture market share. Microsoft and Intel promoted USB, while Firewire came out of Apple. Today, both these serial interfaces have become industry standards for connecting both slow-speed and high-speed peripherals. USB 1.0 could handle transfer rates up to 1.5 MB/sec and was typically used for aggregating the I/O of slow-speed devices such as keyboard and mouse. Firewire could support data transfers at speeds up to a 100 MB/sec and was typically used for multimedia consumer electronics devices such as digital camcorders. USB 2.0 supports data rates up to 60 MB/sec and thus the distinction between Firewire and USB is getting a bit blurred.

To complete the discussion of I/O buses, we should mention two more enhancements to the I/O architecture that are specific to the PC industry. *AGP*, which stands for *Advanced Graphics Port*, is a specialized dedicated channel for connecting 3-D graphics controller card to the motherboard. Sharing the bandwidth available on the PCI bus with other devices was found inadequate, especially for supporting interactive games that led to the evolution of the AGP channel for 3-D graphics. In recent times, AGP has been largely replaced by *PCI Express* (*PCI-e*), which is a new standard that also provides a dedicated connection between the motherboard and graphics controller. Detailed discussions of the electrical properties of these standards and the finer differences among them are beyond the scope of this book.

10.11.1 Dynamic Loading of Device Drivers

Just as the devices are plug and play, so are the device drivers for controlling them. Consider for a moment the device driver for a digital camera. The driver for this device need not be part of your system software (see Figure 10.9) all the time. In modern operating systems such as Linux and Microsoft Vista, the device drivers get "dynamically" linked into the system software when the corresponding device comes on line. The operating system recognizes (via device interrupt), when a new device comes on line (e.g., when you plug in your camera or flash memory into a USB port). The operating system looks through its list of device drivers and identifies the one that corresponds to the hardware that came on line (most likely the device vendor in cooperation with the operating system vendor developed the device driver). It then dynamically links and loads the device driver into memory for controlling the device. Of course, if the device plugged in does not have a matching driver, then the operating system cannot do much except to ask the user to provide a driver for the new hardware device plugged into the system.

10.11.2 Putting it all Together

With the advent of such high-level interfaces to the computer system, the devices be they slow-speed or high-speed are getting farther and farther away from the CPU itself. Thus programmed I/O is almost a thing of the past. In Section 10.5, we mentioned the IBM innovation in terms of I/O processors in the context of mainframes in the 60's and 70's. Now such concepts have made their way into your PC.

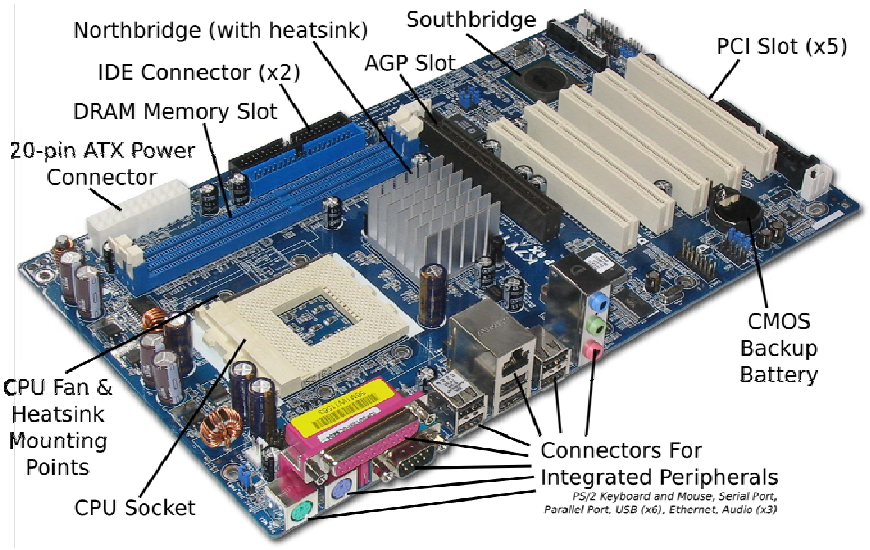


Figure 10.24: Picture of a Motherboard (ASRock K7VT4A Pro)[15]

Motherboard is a term coined in the PC era to signify the circuitry that is central to the computer system. It is single printed circuit board consisting of the processor, memory system (including the memory controller), and the I/O controller for connecting the peripheral devices to the CPU. The name comes from the fact that the printed circuit board contains slots into which expansion boards (usually referred to as *daughter* cards) may be plugged in to expand the capabilities of the computer system. For example, expansion of physical memory is accomplished in this manner. Figure 10.24 shows a picture of a modern motherboard. The picture is self-explanatory in terms of the components and their capabilities. The picture shows the slots into which daughter cards for peripheral controllers and DIMMS (see Chapter 9 for discussion of DIMMS) may be plugged in.

Figure 10.25 shows the block diagram of the important circuit elements inside a modern motherboard. It is worth understanding some of these elements. Every computer system needs some low-level code that executes automatically upon power up. As we already know, the processor simply executes instructions. The trick is to bring the computer system to the state where the operating system is in control of all the resources. You have heard the term *booting* up the operating system. The term is short for *bootstrapping*, an allusion to picking oneself up using the bootstraps of your shoes. Upon power up, the processor automatically starts executing a bootstrapping code that is in a well-known fixed location in read-only memory (ROM). This code does all the initialization of the system including the peripheral devices before transferring the control to the upper layers of the system software shown in Figure 10.9. In the world of PCs, this bootstrapping code is called *BIOS*, which stands for *Basic Input/Output System*.

[15] Picture Courtesy:
http://en.wikipedia.org/wiki/Image:ASRock_K7VT4A_Pro_Mainboard_Labeled_English.png

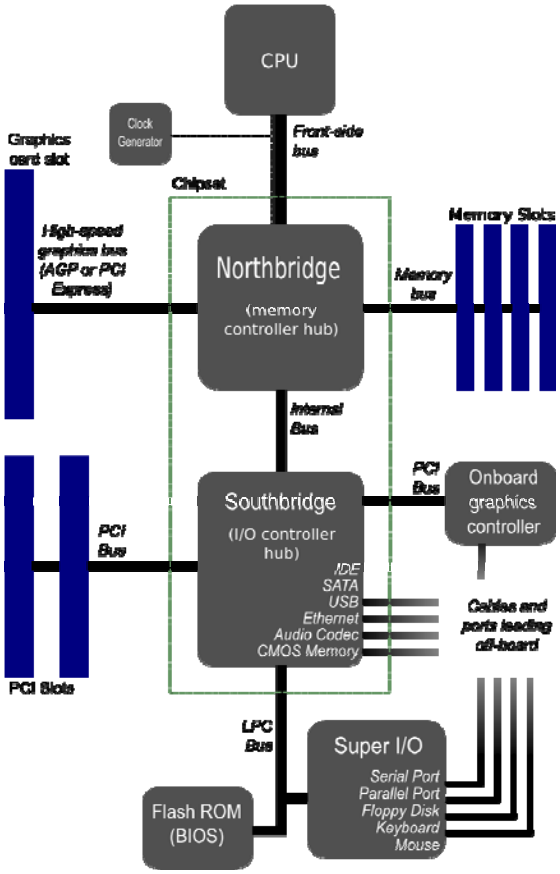


Figure 10.25: Block Diagram of a Motherboard[16]

The following additional points are worth noting with respect to Figure 10.25:

- The box labeled *Northbridge* is a chip that serves as the *hub* for orchestrating the communication between the CPU, and memory system as well as the I/O controllers.
- Similarly, the box labeled *Southbridge* is a chip that serves as the I/O controller hub. It connects to the standard I/O buses that we discussed in this section including PCI and USB, and arbitrates among the devices among the buses and their needs for direct access to the memory via the Northbridge as well as for interrupt service by the CPU. It embodies many of the functions that we discussed in the context of an I/O processor in Section 10.5.
- PCI Express is another bus standard that supports the high transfer rate and response time needs of devices such as a high resolution graphics display.
- LPC, which stands for *Low Pin Count* bus is another standard for connecting low bandwidth devices (such as keyboard and mouse) to the CPU.
- The box labeled *Super I/O* is a chip that serves the I/O controller for a number of slow-speed devices including keyboard, mouse, and printer.

[16] Picture courtesy: http://en.wikipedia.org/wiki/Image:Motherboard_diagram.png

As can be seen from this discussion, the hardware inside the computer system is quite fascinating. While we have used the PC as a concrete example in this section, the functionalities embodied in the elements shown in Figure 10.25 generalizes to any computer system. There was a time when the inside of a computer system would be drastically different depending on the class of machine ranging from a personal computer such as an IBM PC to a Vector Supercomputer such as a Cray-1. With advances in single chip microprocessor technology that we discussed in Chapter 5, and the level of integration made possible by Moore's law (see Chapter 3), there is a commonality in the building blocks used to assemble computer systems ranging from PCs, to desktops, to servers, to supercomputers.

10.12 Summary

In this chapter, we covered the following topics:

1. Mechanisms for communication between the processor and I/O devices including programmed I/O and DMA.
2. Device controllers and device drivers.
3. Buses in general and evolution of I/O buses in particular, found in modern computer systems.
4. Disk storage and disk-scheduling algorithms.

In the next chapter, we will study file systems, which is the software subsystem built on top of stable storage in general, hard disk in particular.

10.13 Review Questions

1. Compare and contrast program controlled I/O with Direct Memory Access (DMA).

2. Given a disk drive with the following characteristics:
   - Number of surfaces = 200
   - Number of tracks per surface = 100
   - Number of sectors per track = 50
   - Bytes per sector = 256
   - Speed = 2400 RPM

   What is the total disk capacity?

   What is the average rotational latency?

3. A disk has 20 surfaces (i.e. 10 double sided platters). Each surface has 1000 tracks. Each track has 128 sectors. Each sector can hold 64 bytes. The disk space allocation policy allocates an integral number of contiguous cylinders to each file,

   How many bytes are contained in one cylinder?

   How many cylinders are needed to accommodate a 5-Megabyte file on the disk?

   How much space is wasted in allocating this 5-Megabyte file?

4. A disk has the following configuration:

   The disk has 310 MB
   Track size: 4096 bytes
   Sector Size: 64 bytes

   A programmer has 96 objects each being 50 bytes in size. He decides to save each object as an individual file. How many bytes (in total) are actually written to disk?

5. Describe in detail the sequence of operations involved in a DMA data transfer.

6. What are the mechanical things that must happen before a disk drive can read data?

7. A disk drive has 3 double-sided platters. The drive has 300 cylinders. How many tracks are there?

8. While a disk unit is performing a DMA transfer, what device minimizes the number of bus cycles that the processor must steal from the DMA transfer?

9. What is the normal objective of a disk scheduling algorithm?

10. Using the number of tracks traversed as a measure of the time; compare all the disk scheduling algorithms for the specifics of the disk and the request pattern given in Example 4, with respect to the different performance metrics summarized in Table 10.3.

11. Assume the same details about the disk as in Example 4. The request queue does not remain the same but it changes as new requests are added to the queue. At any point of time the algorithm uses the current requests to make its decision as to the next request to be serviced. Consider the following request sequence:
    - Initially (at time 0) the queue contains requests for cylinders: 99, 3, 25
    - At the time the next decision has to be taken by the algorithm, one new requests has joined the queue: 46
    - Next decision point one more request has joined the queue: 75
    - Next decision point one more request has joined the queue: 55
    - Next decision point one more request has joined the queue: 85
    - Next decision point one more request has joined the queue: 73
    - Next decision point one more request has joined the queue: 50

    Assume that the disk head at time 0 is just completing a request at track 55.

    a) Show the schedule for FCFS, SSTF, SCAN, C-SCAN, LOOK, and C-LOOK.
    b) Compute the response time (in units of head traversal) for each of the above requests.

c) What is the average response time and throughput observed for each of the algorithms?