Module 5: Input Output (IO) Management

So far we have studied how resources like processor and main memory are managed. We shall now examine the I/O management. Humans interact with machines by providing information through IO devices. Also, much of whatever a computer system provides as on-line services is essentially made available through specialized devices such as screen displays, printers, keyboards, mouse, etc. Clearly, management of all these devices can affect the throughput of a system. For this reason, input output management also becomes one of the primary responsibilities of an operating system. In this chapter we shall examine the role of operating systems in managing IO devices. In particular, we shall examine how the end use of the devices determines the way they are regulated for communication with either humans or with systems.

5.1 Issues in IO Management

Let us first examine the context of input output in a computer system. We shall look at issues initially from the point of view of communication with a device. Later, in Section 5.1.1, we shall also examine issues from the point of view of managing events. When we analyze device communication, we notice that communication is required at the following three levels:

- The need for a human to input information and receive output from a computer.
- The need for a device to input information and receive output from a computer.
- The need for computers to communicate (receive/send information) over networks.

The first kind of IO devices operate at rates good for humans to interact. These may be character-oriented devices like a keyboard or an event-generating device like a mouse. Usually, human input using a key board will be a few key depressions at a time. This means that the communication is rarely more than a few bytes. Also, the mouse events can be encoded by a small amount of information (just a few bytes). Even though a human input is very small, it is stipulated that it is very important, and therefore requires an immediate response from the system. A communication which attempts to draw attention often requires the use of an interrupt mechanism or a programmed data mode of operation. Interrupt as well as programmed data mode of IO shall be dealt with in detail later in this chapter.

The second kind of IO requirement arises from devices which have a very high character density such as tapes and disks. With these characteristics, it is not possible to regulate communication with devices on a character by character basis. The information transfer, therefore, is regulated in blocks of information. Additionally, sometimes this may require some kind of format control to structure the information to suit the device and/or data characteristics. For instance, a disk drive differs from a line printer or an image scanner. For each of these devices, the format and structure of information is different. It should be observed that the rate at which a device may provide data and the rates at which an end application may consume it may be considerably different. In spite of these differences, the OS should provide uniform and easy to use IO mechanisms. Usually, this is done by providing a buffer. The OS manages this buffer so as to be able to comply with the requirements of both the producer and consumer of data. In section 5.4 we discuss the methods to determine buffer sizes.

The third kind of IO requirements emanate from the need to negotiate system IO with the communications infrastructure. The system should be able to manage communications traffic across the network. This form of IO facilitates access to internet resources to support e-mail, file-transfer amongst machines or Web applications. Additionally now we have a large variety of options available as access devices. These access devices may be in the form of Personal Digital Assistant (PDA), or mobile phones which have infrared or wireless enabled communications. This rapidly evolving technology makes these forms of communications very challenging. It is beyond the scope of this book to have a discussion on these technologies, devices or mechanisms. Even then it should be remarked that most network cards are direct memory access (DMA) enabled to facilitate DMA mode of IO with communication infrastructure. We shall discuss DMA in Section 5.2.5. Typically the character-oriented devices operate with speeds of tens of bytes per second (for keyboards, voice-based input, mouse, etc.). The second kind of devices operate over a much wider range. Printers operate at 1 to 2 KB per second, disks transfer at rates of 1 MB per second or more. The graphics devices fall between these two ranges while the graphics cards may in fact be even faster. The devices communicate with a machine using a data bus and a device controller. Essentially, all these devices communicate large data blocks. However, the communication with networks differs from the way the communication takes place with the block devices. Additionally, the

communication with wireless devices may differ from that required for internet services. Each of these cases has its own information management requirements and OS must negotiate with the medium to communicate. Therefore, the nature of the medium controls the nature of protocol which may be used to support the needed communication.

One of the important classes of OS is called Real-time Operating Systems or RTOS for short. We shall study RTOS in a later chapter. For now, let us briefly see what distinguishes an RTOS from a general purpose OS. RTOSs are employed to regulate a process and generate responses to events in its application environments within a stipulated time considered to be real-time response time. RTOS may be employed to regulate a process or even offer transaction oriented services like on-line reservation system, etc. The main point of our concern here is to recognize the occurrence of certain events or event order. A key characteristic of embedded systems is to recognize occurrence of events which may be by monitoring variable values or identifying an event. For now let us study the IO management.

5.1.1 Managing Events

Our next observation is that a computer system may sometimes be embedded to interact with a real-life activity or process. It is quite possible that in some operational context a process may have to synchronize with some other process. In such a case this process may actually have to wait to achieve a rendezvous with another process. In fact, whichever of the two synchronizing processes arrives first at the point of rendezvous would have to wait. When the other process also reaches the point of synchronization, the first process may proceed after recognizing the synchronizing event. Note that events may be communicated using signals which we shall learn about later.

In some other cases a process may have to respond to an asynchronous event that may occur at any time. Usually, an asynchronous input is attended to in the next instruction cycle as we saw in Section 1.2. In fact, the OS checks for any event which may have occurred in the intervening period. This means that an OS incorporates some IO event recognition mechanism. IO handling mechanisms may be like polling, or a programmed data transfer, or an interrupt mechanism, or even may use a direct memory access (DMA) with cycle stealing. We shall examine all these mechanisms in some detail in Section 5.2.

The unit of data transfer may either be one character at a time or a block of characters. It may require to set up a procedure or a protocol. This is particularly the case when a

machine-to-machine or a process-to-process communication is required. Additionally, in these cases, we need to account for the kind of errors that may occur. We also need procedure to recover when such an error occurs. We also need to find ways to ensure the security and protection of information when it is in transit. Yet another important consideration in protection arises when systems have to share devices like printers. We shall deal with some of these concerns in the next module on resource sharing. In the discussions above we have identified many issues in IO. For now let us look at how IO mechanisms are organized and how they operate.

5.2 IO Organization

In the previous section we discussed various issues that arise from the need to support a wide range of devices. To meet these varied requirements, a few well understood modalities have evolved over time. The basic idea is to select a mode of communication taking device characteristics into account or a need to synchronize with some event, or to just have a simple strategy to ensure a reliable assured IO.

Computers employ the following four basic modes of IO operation:

1. Programmed mode
2. Polling mode
3. Interrupt mode
4. Direct memory access mode.

We shall discuss each of these modes in some detail now.

5.2.1 Programmed Data Mode

In this mode of communication, execution of an IO instruction ensures that a program shall not advance till it is completed. To that extent one is assured that IO happens before anything else happens. As depicted in Figure 5.1, in this mode an IO instruction is issued to an IO device and the program executes in “busy-waiting” (idling) mode till the IO is completed. During the busy-wait period the processor is continually interrogating to check if the device has completed IO. Invariably the data transfer is accomplished through an identified register and a flag in a processor. For example, in Figure 5.1 depicts

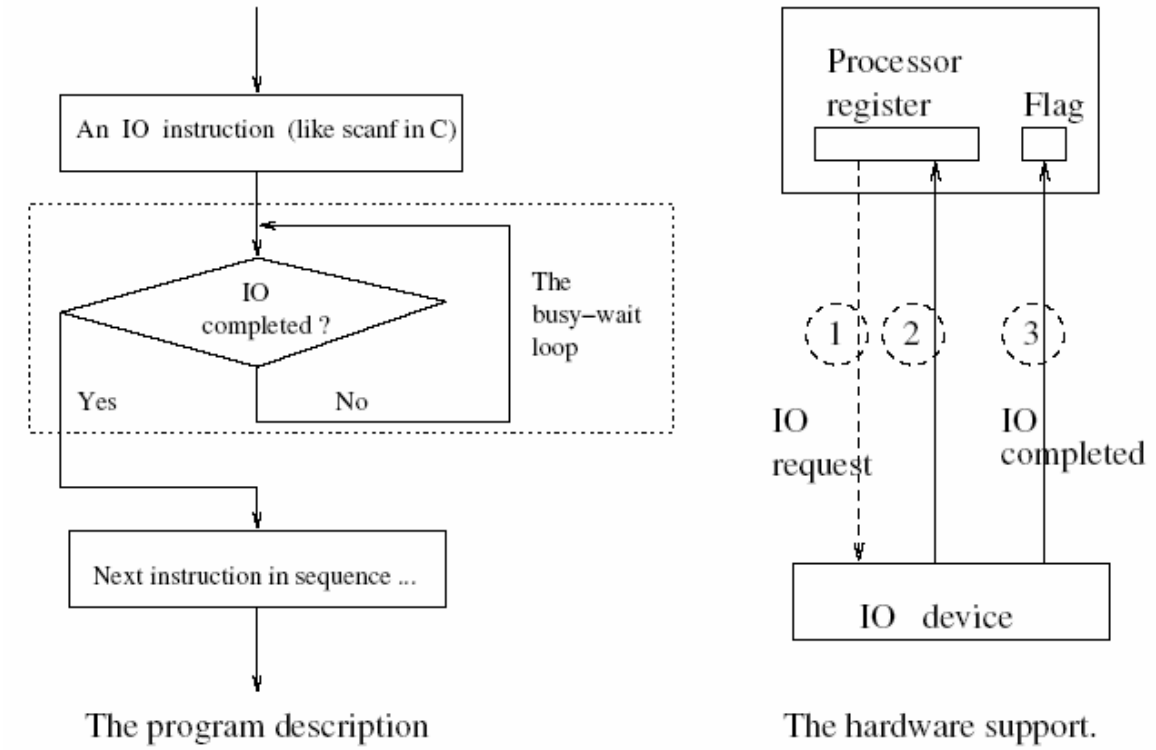

Figure 5.1: Programmed mode of IO.

how an input can happen using the programmed data mode. First the processor issues an IO request (shown as 1), followed by device putting a data in a register (shown as 2) and finally the flag (which is being interrogated) is set (shown as 3). The device either puts a data in the register (as in case of input) or it picks up data from the register (in case of output). When the IO is accomplished it signals the processor through the flag. During the busy-wait period the processor is busy checking the flag. However, the processor is idling from the point of view of doing anything useful. This situation is similar to a car engine which is running when the car is not in motion – essentially "idling".

5.2.2 Polling

In this mode of data transfer, shown in Figure 5.2, the system interrogates each device in turn to determine if it is ready to communicate. If it is ready, communication is initiated and subsequently the process continues again to interrogate in the same sequence. This is just like a round-robin strategy. Each IO device gets an opportunity to establish

Communication in turn. No device has a particular advantage (like say a priority) over other devices.

Polling is quite commonly used by systems to interrogate ports on a network. Polling may also be scheduled to interrogate at some pre-assigned time intervals. It should be

remarked here that most daemon software operate in polling mode. Essentially, they use a while true loop as shown in Figure 5.2.

In hardware, this may typically translate to the following protocol:

1. Assign a distinct address to each device connected to a bus.
2. The bus controller scans through the addresses in sequence to find which device wishes to establish a communication.
3. Allow the device that is ready to communicate to leave its data on the register.
4. The IO is accomplished. In case of an input the processor picks up the data. In case of an output the device picks up the data.
5. Move to interrogate the next device address in sequence to check if it is ready to communicate.

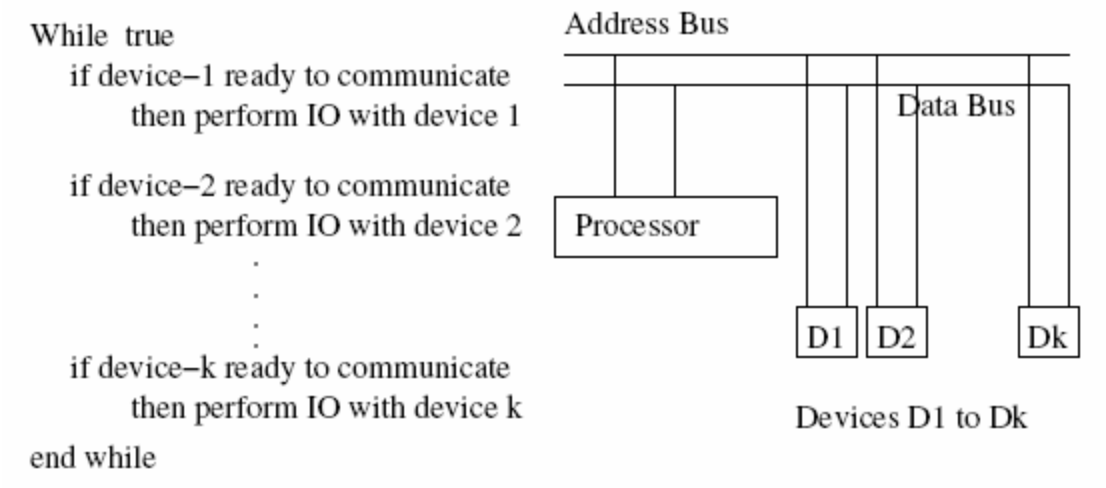

Figure 5.2: Polling mode of IO.

As we shall see next, polling may also be used within an interrupt service mode to identify the device which may have raised an interrupt.

5.2.3 Interrupt Mode

Let us begin with a simple illustration to explain the basic rationale behind interrupt mode of data transfer. Suppose a program needs input from a device which communicates using interrupt. Even with the present-day technology the devices are one thousand or more times slower than the processor. So if the program waits on the input device it would cycle through many processor cycles just waiting for the input device to be ready to communicate. This is where the interrupt mode of communication scores.

To begin with, a program may initiate IO request and advance without suspending its operation. At the time when the device is actually ready to establish an IO, the device

raises an interrupt to seek communication. Immediately the program execution is suspended temporarily and current state of the process is stored. The control is passed on to an interrupt service routine (which may be specific to the device) to perform the desired input. Subsequently, the suspended process context is restored to resume the program from the point of its suspension.

Interrupt processing may happen in the following contexts:

- **Internal Interrupt**: The source of interrupt may be a memory resident process or a function from within the processor. We regard such an interrupt as an internal interrupt. A processor malfunction results in an internal interrupt. An attempt to divide by zero or execute an illegal or non-existent instruction code results in an internal interrupt as well. A malfunction arising from a division by zero is called a trap. Internal interrupt may be caused by a timer as well. This may be because either the allocated processor time slice to a process has elapsed or for some reason the process needs to be pre-empted. Note that an RTOS may pre-empt a running process by using an interrupt to ensure that the stipulated response time required is met. This would also be a case of internal interrupt.
- **External Interrupt:** If the source of interrupt in not internal, i.e. it is other than a process or processor related event then it is an external interrupt. This may be caused by a device which is seeking attention of a processor. As indicated earlier, a program may seek an IO and issue an IO command but proceed. After a while, the device from which IO was sought is ready to communicate. In that case the device may raise an interrupt. This would be a case of an external interrupt.
- **Software Interrupt:** Most OSs offer two modes of operation, the user mode and the system mode. Whenever a user program makes a system call, be it for IO or a special service, the operation must have a transition from user mode to system mode. An interrupt is raised to effect this transition from user to system mode of operation. Such an interrupt is called a software interrupt.

We shall next examine how an interrupt is serviced. Suppose we are executing an instruction at *i* in program P when interrupt signal has been raised. Let us also assume that we have an interrupt service routine which is to be initiated to service the interrupt. The following steps describe how a typical interrupt service may happen.

- Suspend the current program P after executing instruction i.

- Store the address of instruction at *i + 1* in P as the return address. Let us denote this address as $PADDR_{i+1}$. This is the point at which program P shall resume its execution following the interrupt service. The return address is essentially the incremented program counter value. This may be stored either in some specific location or in some data structure (like a stack or an array). The transfer of control to an interrupt service routine may also be processed like a call to a subroutine. In that case, the return address may even be stored in the code area of the interrupt service routine. Let us identify the location where we stored $PADDR_{i+1}$ as the address *RESTORE*. Later, in step-4, we shall see how storing the return address helps to restore the original sequence of program starting at $PADDR_{i+1}$.
- Execute a branch unconditionally to transfer control to the interrupt service instructions. The immediately following instruction cycle initiates the interrupt service routine.
- Typically the last instruction in the service routine executes a branch indirect from the location RESTORE. This restores the program counter to take the next instruction at $PADDR_{i+1}$. Thus the suspended program P obtains the control of the processor again

5.2.4 Issues in Handling Interrupts

There are many subtle issues and points that need clarification. We shall examine some of these in some detail next.

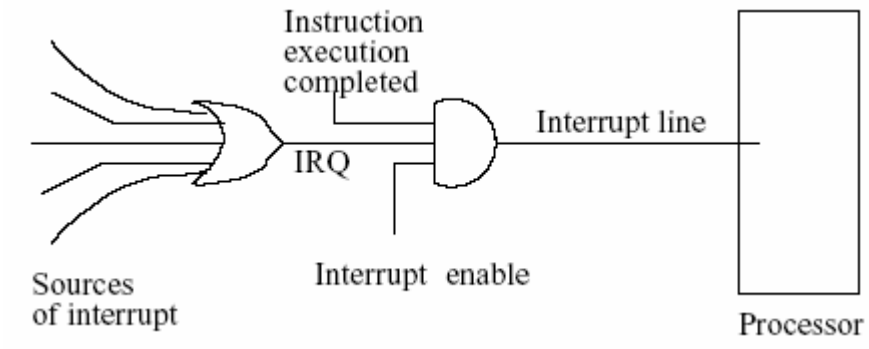

Figure 5.3: How an interrupt is detected.

- **Detecting an interrupt**: In Figure 5.3 we depict how a processor may detect an interrupt request. When a processor has an interrupt enable signal up, then at the end of an instruction cycle we shall recognize an interrupt if the interrupt request (IRQ) line is up. Once an interrupt is recognized, interrupt service shall be

required. Two minor points now arise. One is when is interrupt enabled. The other is how a device which raises the interrupt is identified.

- **When is interrupt enabled:** Sources of interrupt may have assigned priorities. If a higher priority device seeks an interrupt, the lower priority device may be denied the per-mission to raise an interrupt. Also, in many real-time systems there may be a critical operation which must proceed without an interrupt. Later, in the next chapter, we will see that certain system operations, particularly which deal with resource allocation or semaphores, cannot be interrupted. In that case a processor may disable interrupt. Next we will look at the mechanisms that may be used to identify the source of an interrupt. Yet another simple mechanism would be to use a mask register. By setting a mask, one can disable some interrupt request lines to send interrupt requests.
- **Identifying the source of interrupt:** As we saw in Figure 5.3 many devices that wish to raise interrupt are gated in via an OR gate to raise an interrupt request (IRQ). Usually, the requests from a set of devices with equal priority would be pooled. Such a pool may have a set of devices under a cluster controller or an IO processor of some kind. One way of determining is to use polling which we discussed earlier. Another mechanism may be to have a daisy-chain arrangement of devices as shown in Figure 5.4. We notice that the normally closed position of the switch allows interrupt to be raised from the devices lower down in the chain.

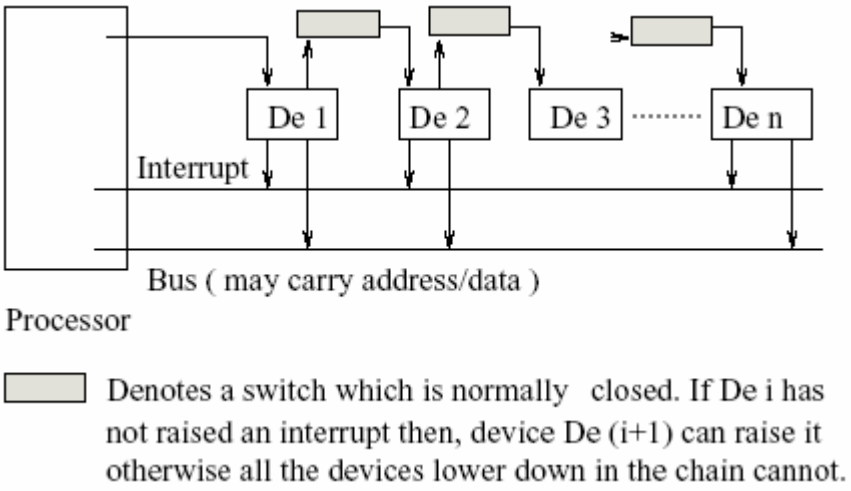

Figure 5.4: Daisy chained devices.

The interrupting device raises the interrupt and the address or data may then be asserted on the bus as shown in Figure 5.4.

It is also possible that there may be an interrupt raised through an IO cluster (which may be for a SCSI device). In that case there would have to be a small hardware in which the specific address of the device as well as data may be stored. A small protocol would then resolve the IO.

- **Interrupt received when a different process is executing:** Suppose the process Pi initiated an IO. But subsequently it relinquishes the processor to process Pj . This may well happen because the process Pi may have finished its time slice or it may have reached a point when it must process a data expected from the device. Now let us suppose that priority of process Pi is higher than that of Pj. In that case the process Pj shall be suspended and Pi gets the processor, and the IO is accomplished. Else the process Pj shall continue and the IO waits till the process Pi is able to get the control of the processor. One other way would be to let the device interrupt the process Pj but store the IO information in a temporary buffer (which may be a register) and proceed. The process Pj continues. Eventually, the process Pi obtains the CPU. It would pick up the data from the temporary buffer storage. Clearly, the OS must provide for such buffer management. We next examine nested interrupts i.e. the need to service an interrupt which may occur during an interrupt service.
- **An Interrupt during an interrupt service:** This is an interesting possibility. Often devices or processes may have priorities. A lower priority process or device cannot cause an interrupt while a higher priority process is being serviced. If, however, the process seeking to interrupt is of higher priority then we need to service the interrupt.

  In the case where we have the return address deposited in the interrupt service routine code area, this can be handled exactly as the nested subroutine calls are handled. The most nested call is processed first and returns to the address stored in its code area. This shall always transfer the control to next outer layer of call. In the case we have a fixed number of interrupt levels, the OS may even use a stack to store the return addresses and other relevant information which is needed.
- **Interrupt vector:** Many systems support an interrupt vector (IV). As depicted in

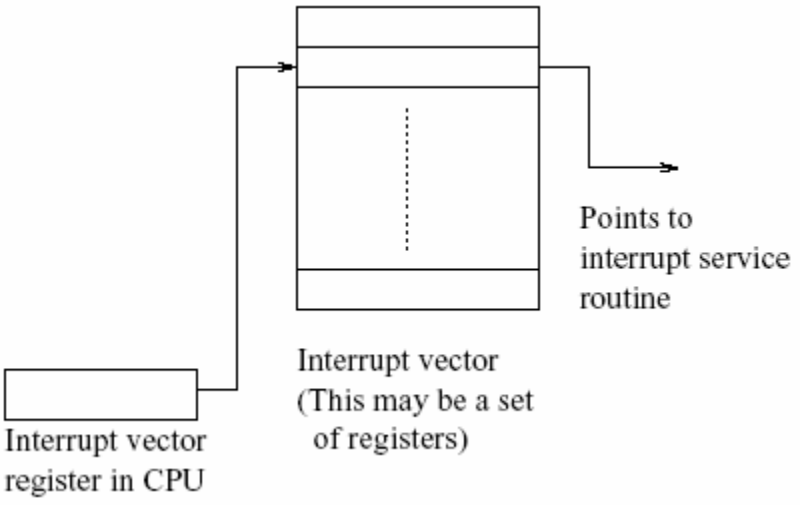

Figure 5.5: Interrupt vectors.

Figure 5.5, the basic idea revolves around an array of pointers to various interrupt service routines. Let us consider an example with four sources of interrupt. These may be a trap, a system call, an IO, or an interrupt initiated by a program. Now we may associate an index value 0 with trap, 1 with system call, 2 with IO device and 3 with the program interrupt. Note that the source of interrupt provides us the index in the vector. The interrupt service can now be provided as follows:

- Identify the source of interrupt and generate index i.
- Identify the interrupt service routine address by looking up IVR(i), where IVR stands for the interrupt vector register. Let this address be ISRi.
- Transfer control to the interrupt service routine by setting the program counter to ISRi.

Note that the interrupt vector may also be utilized in the context of a priority-based interrupt in which the bit set in a bit vector determines the interrupt service routine to be selected. It is very easy to implement this in hardware.

5.2.5 DMA Mode of Data Transfer

This is a mode of data transfer in which IO is performed in large data blocks. For instance, the disks communicate in data blocks of sizes like 512 bytes or 1024 bytes. The direct memory access, or DMA ensures access to main memory without processor intervention or support. Such independence from processor makes this mode of transfer extremely efficient.

When a process initiates a direct memory access (DMA) transfer, its execution is briefly suspended (using an interrupt) to set up the DMA control. The DMA control requires the information on starting address in main memory and size of data for transfer. This information is stored in DMA controller. Following the DMA set up, the program resumes from the point of suspension. The device communicates with main memory stealing memory access cycles in competition with other devices and processor. Figure 5.6 shows the hardware support.

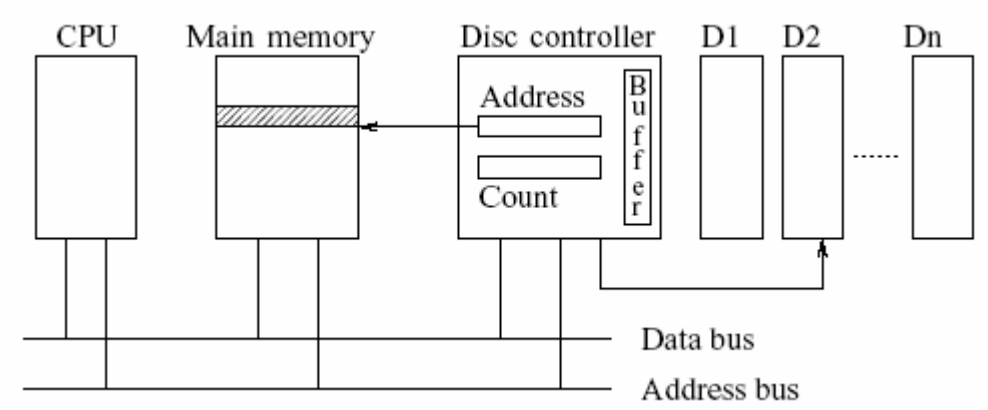

Figure 5.6: DMA : Hardware support.

Let us briefly describe the operations shown in Figure 5.6. Also, we shall assume a case of disk to main memory transfer in DMA mode. We first note that there is a disk controller to regulate communication from one or more disk drives. This controller essentially isolates individual devices from direct communication with the CPU or main memory. The communication is regulated to first happen between the device and the controller, and later between the controller and main memory or CPU if so needed. Note that these devices communicate in blocks of bits or bytes as a data stream. Clearly, an un-buffered communication is infeasible via the data bus. The bus has its own timing control protocol. The bus cannot, and should not, be tied to device transfer bursts. The byte stream block needs to be stored in a buffer isolated from the communication to processor or main memory. This is precisely what the buffer in the disk controller accomplishes. Once the controller buffer has the required data, then one can envisage to put the controller in contention with CPU and main memory or CPU to obtain an access to the bus. Thus if the controller can get the bus then by using the address and data bus it can directly communicate with main memory. This transfer shall be completely independent of program control from the processor. So we can effect a transfer of one block of data

from the controller to main memory provided the controller has the address where data needs to be transferred and data count of the transfer required. This is the kind of information which initially needs to be set up in the controller address and count registers. Putting this information may be done under a program control as a part of DMA set up. The program that does it is usually the device controller. The device controller can then schedule the operations with much finer control. Data location information in disk

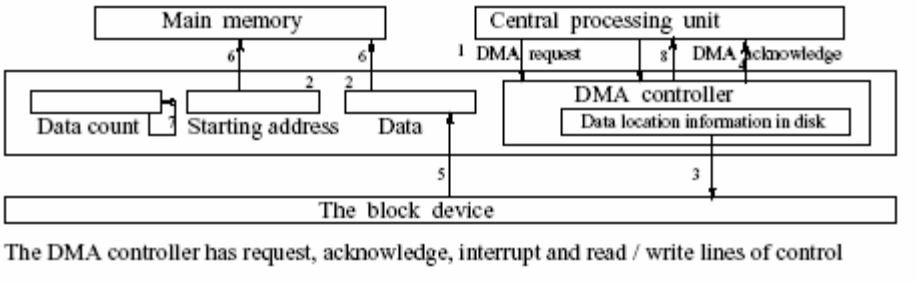

Figure 5.7: DMA : Direct memory access mode of data transfer.

Let us now recap the above mode of transfer within a logical framework with a step-by-step description. This procedure can be understood in the context of Figure 5.7. In the figure a few lines do not have an accompanying label. However, by following the numbered sequence and its corresponding label, one can find the nature of the operations considered.

The procedure can be understood by following the description given below:

We shall assume that we have a program P which needs to communicate with a device D and get a chunk of data D to be finally stored starting in address A. The reader should follow through the steps by correlating the numbers in Figure 5.7.

1. The program makes a DMA set-up request.
2. The program deposits the address value A and the data count D.the program also indicates the virtual memory address of the data on disk.
3. The DMA controller records the receipt of relevant information and acknowledges the DMA complete.
4. The device communicates the data to the controller buffer.
5. The controller grabs the address bus and data bus to store the data, one word at a time.
6. The data count is decremented.

7. The above cycle is repeated till the desired data transfer is accomplished. At which time a DMA data transfer complete signal is sent to the process.

The network-oriented traffic (between machines) may be handled in DMA mode. This is so because the network cards are often DMA enabled. Besides, the network traffic usually corresponds to getting information from a disk file at both the ends. Also, because network traffic is in bursts, i.e. there are short intervals of large data transfers. DMA is the most preferred mode of communication to support network traffic.

5.2.6 A Few Additional Remarks

In every one of the above modes of device communication, it must be remarked that the OS makes it look as if we are doing a read or a write operation on a file. In the next section we explore how this illusion of a look and feel of a file is created to effect device communication. Also note that we may have programmed IO for synchronizing information between processes or when speed is not critical. For instance, a process may be waiting for some critical input information required to advance the computation further. As an example of programmed IO, we may consider the PC architecture based on i386 CPU which has a notion of listening to an IO port. Some architectures may even support polling a set of ports. The interrupt transfer is ideally suited for a small amount of critical information like a word, or a line i.e. no more than tens of bytes.

5.3 HW/SW Interface

IO management requires that a proper set-up is created by an application on computer system with an IO device. An IO operation is a combination of HW and SW instructions as shown in Figure 5.8.

Following the issuance of an IO command, OS kernel resolves it, and then communicates

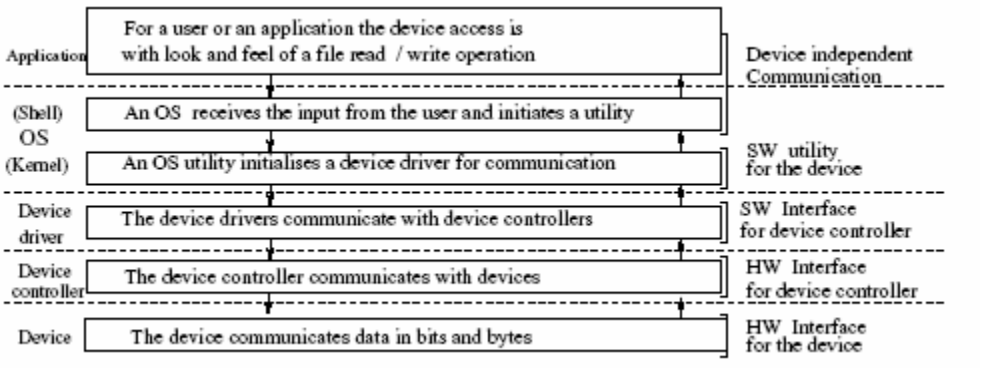

Figure 5.8: Communication with IO devices.

with the concerned device driver. The device drivers in turn communicate with IO devices.

The application at the top level only communicates with the kernel. Each IO request from an application results in generating the following:

- Naming or identification of the device to communicate.
- Providing device independent data to communicate;

The kernel IO subsystem arranges for the following:

- The identification of the relevant device driver. We discuss device drivers in Section 5.3.1.
- Allocation of buffers. We discuss buffer management in Section 5.4.
- Reporting of errors.
- Tracking the device usage (is the device free, who is the current user, etc.) The device driver transfers a kernel IO request to set up the device controller. A device controller typically requires the following information:
- Nature of request: read/write.
- Set data and control registers for data transfer (initial data count = 0; where to look for data, how much data to transfer, etc.)
- Keep track when the data has been transferred (when fresh data is to be brought in). This may require setting flags.

5.3.1 Device Drivers

A device driver is a specialized software. It is specifically written to manage communication with an identified class of devices. For instance, a device driver is specially written for a printer with known characteristics. Different make of devices may differ in some respect, and therefore, shall require different drivers. More specifically, the devices of different makes may differ in speed, the sizes of buffer and the interface characteristics, etc. Nevertheless device drivers present a uniform interface to the OS. This is so even while managing to communicate with the devices which have different characteristics.

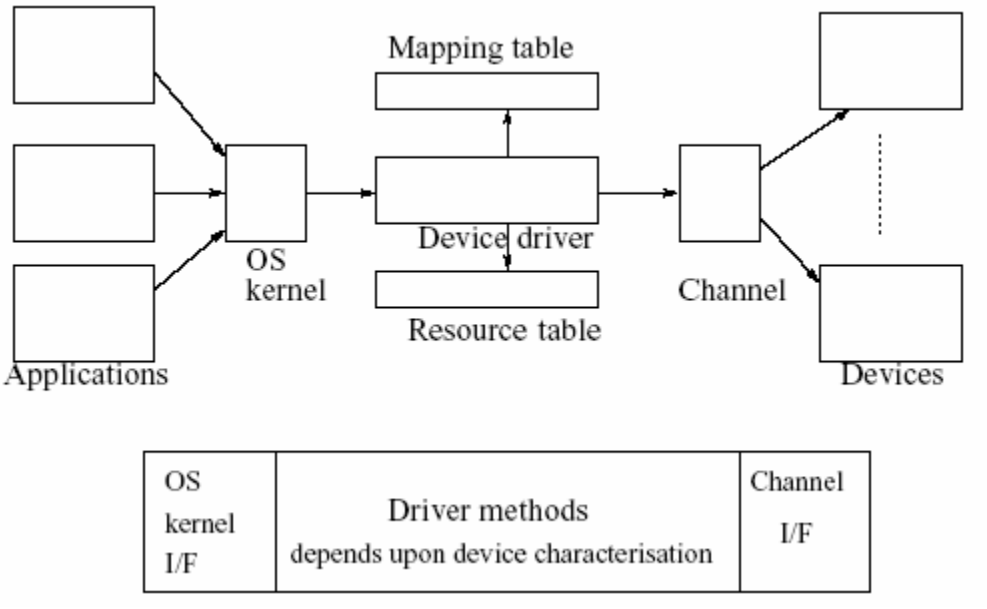

Figure 5.9: Device-driver interface.

In a general scenario, as shown in Figure 5.9, n applications may communicate with m devices using a common device driver. In that case the device driver employs a mapping table to achieve communication with a specific device. Drivers may also need to use specific resources (like a shared bus). If more than one resource is required, a device driver may also use a resource table. Sometimes a device driver may need to block a certain resource for its exclusive use by using a semaphore 1 .

The device driver methods usually have device specific functionalities supported through standard function calls. Typically, the following function calls are supported. *open(), close(), lseek(), read(), write ()*

These calls may even be transcribed as *hd-open(), hd-close(),* etc. to reflect their use in the context of hard-disk operations. Clearly, each driver has a code specific to the device (or device controller). Semantically, the user views device communication as if it were a communication with a file. Therefore, he may choose to transfer an arbitrary amount of data. The device driver, on the other hand, has to be device specific. It cannot choose an arbitrary sized data transfer. The driver must manage fixed sizes of data for each data transfer. Also, as we shall see during the discussion on buffer transfer in Section 5.4, it is an art to decide on the buffer size. Apparently, with n applications communicating with m devices the device driver methods assume greater levels of complexity in buffer management.

Sometimes the device drives are written to emulate a device on different hardware. For instance, one may emulate a RAM-disk or a fax-modem. In these cases, the hard-ware (on which the device is emulated) is made to appear like the device being emulated. The call to seek service from a device driver is usually a system call as device driver methods are often OS resident. In some cases where a computer system is employed to handle IO exclusively, the device drivers may be resident in the IO processor. In those cases, the communication to the IO processor is done in kernel mode. As a good design practice device drivers may be used to establish any form of communication, be it interrupt or DMA. The next section examines use of a device driver support for interrupt-based input.

5.3.2 Handling Interrupt Using Device Drivers

Let us assume we have a user process which seeks to communicate with an input device using a device driver process. Processes communicate by signaling. The steps in figure 5.10 describe the complete operational sequence (with corresponding numbers).

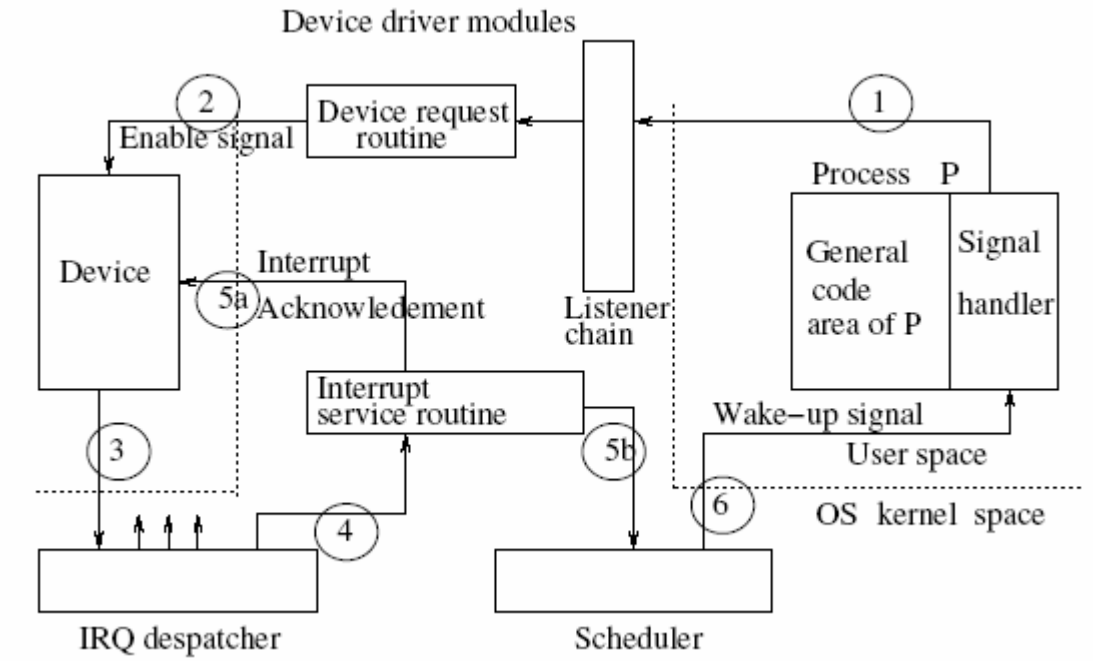

Figure 5.10: Device-driver operation.

1. Register with listener chain of the driver: The user process P signals the device driver as process DD to register its IO request. Process DD maintains a list data structure, basically a listener chain, in which it registers requests received from processes which seek to communicate with the input device.
2. Enable the device: The process DD sends a device enable signal to the device.

3. Interrupt request handling: After a brief while the device is ready to communicate and sends an interrupt request IRQ to process DD. In fact, the interrupt dispatcher in DD ensures that interrupt service is initiated.
4. Interrupt acknowledge and interrupt servicing: The interrupt service routine ISR acknowledges to the device and initiates an interrupt service routine 2 .
5. Schedule to complete the communication: Process DD now schedules the data transfer and follows it up with a wake-up signal to P. The process receives the data to complete the input.
6. Generate a wake up signal.

Just as we illustrated an example of use of a device driver to handle an input device, we could think of other devices as well. One of the more challenging tasks is to write device driver for a pool of printers. It is not uncommon to pool print services. A printer requires that jobs fired at the pool are duly scheduled. There may be a dynamic assignment based on the load or there may even be a specific request (color printing on glazed paper for instance) for these printers.

In supporting device drivers for DMA one of the challenges is to manage buffers. In particular, the selection of buffer size is very important. It can very adversely affect the throughput of a system. In the next section we shall study how buffers may be managed.

5.4 Management of Buffers

A key concern, and a major programming issue from the OS point of view, is the management of buffers. The buffers are usually set up in the main memory. Device drivers and the kernel both may access device buffers. Basically, the buffers absorb mismatch in the data transfer rates of processor or memory on one side and device on the other. One key issue in buffer management is buffer-size. How buffer-sizes may be determined can be explained by using a simple analogy. The analogy we use relates to production and distribution for a consumable product like coffee. The scenario, depicted

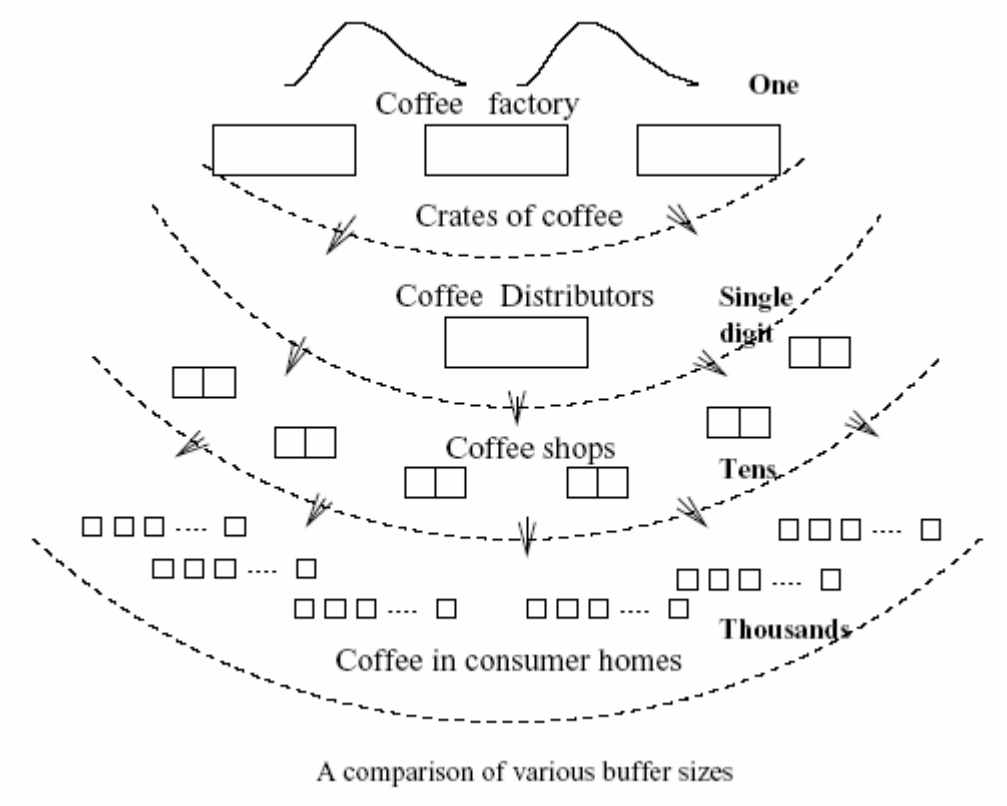

Figure 5.11: Coffee buffers

in Figure 5.11 shows buffer sizes determined by the number of consumers and the rate of consumption. Let us go over this scenario. It is easy to notice that a coffee factory would produce mounds of coffee. However, this is required to be packaged in crates for the distribution. Each crate may hold several boxes or bottles. The distributors collect the crates of boxes in tens or even hundreds for distribution to shops. Now that is buffer management. A super-market may get tens of such boxes while smaller shops like pop-and-mom stores may buy one or possibly two boxes. That is their buffer capacity based on consumption by their clients. The final consumer buys one tin (or a bottle). He actually consumes only a spoonful at one time. We should now look at the numbers and volumes involved. There may be one factory, supplying a single digit of distributors who distribute it to tens of super-markets and / or hundreds of smaller stores. The ultimate consumers number thousands with a very small rate of consumption. Now note that the buffer sizes for factory should meet the demands from a few bulk distributors. The buffer sizes of distributors meet the demand from tens of super-markets and hundreds of smaller shops. The smaller shops need the supplies of coffee bottles at most in tens of bottles (which may be a small part of the capacity of a crate). Finally, the end consumer has a

buffer size of only one bottle. The moral of the story is that at each interface the producers and consumers of commodity balance the demand made on each other by suitably choosing a buffer size. The effort is to minimize the cost of lost business by being out of stock or orders. We can carry this analogy forward to computer operation. Ideally, the buffer sizes should be chosen in computer systems to allow for free flow of data, with neither the producer (process) of data nor the consumer (process) of data is required to wait on the other to make the data available.

Next we shall look at various buffering strategies (see Figure 5.12).

Single buffer: The device first fills out a buffer. Next the device driver hands in its control to the kernel to input the data in the buffer. Once the buffer has been used up, the device fills it up again for input.

Double buffer: In this case there are two buffers. The device fills up one of the two buffers, say buffer-0. The device driver hands in buffer-0 to the kernel to be emptied and the device starts filling up buffer-1 while kernel is using up buffer-0. The roles are switched when the buffer-1 is filled up.

Circular buffer: One can say that the double buffer is a circular queue of size two. We can extend this notion to have several buffers in the circular queue. These buffers are filled up in sequence. The kernel accesses the filled up buffers in the same sequence as these are filled up. The buffers are organized as a circular queue data structure, i.e. in case of output, the buffer is filled up from the CPU(or memory) end and used up by the output device, i.e. buffer n = buffer 0.

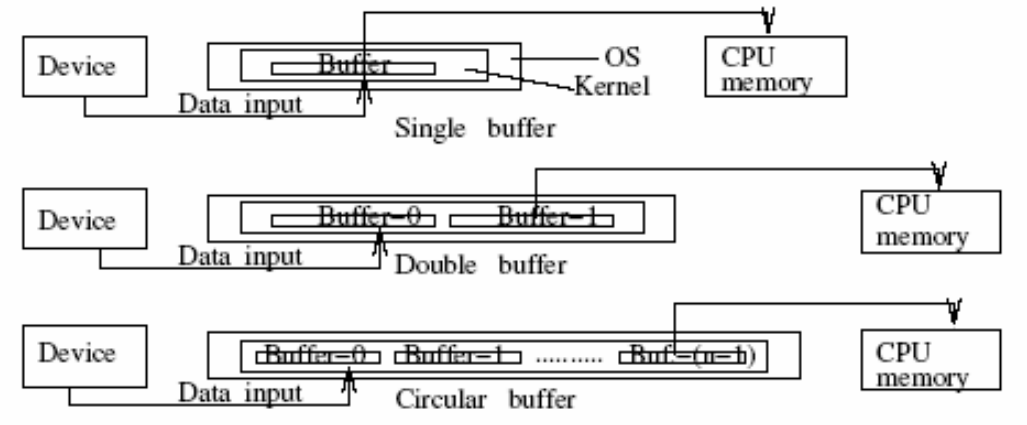

Figure 5.12: Buffering schemes.

Note that buffer management essentially requires managing a queue data structure. The most general of these is the circular buffer. One has to manage the pointers for the queue

head and queue tail to determine if the buffer is full or empty. When not full or not empty the queue data structure can get a data item from a producer or put a data item into the consumer. This is achieved by carefully monitoring the head and tail pointers. A double buffer is a queue of length two and a single buffer is a queue of length one. Before moving on, we would also like to remark that buffer status of full or empty may be communicated amongst the processes as an event as indicated in Section 5.1.1 earlier.

5.5 Some Additional Points

In this section we discuss a few critical services like clocks and spooling. We also discuss many additional points relevant to IO management like caches.

Spooling: Suppose we have a printer connected to a machine. Many users may seek to use the printer. To avoid print clashes, it is important to be able to queue up all the print requests. This is achieved by spooling. The OS maintains all print requests and schedules each users' print requests. In other words, all output commands to print are intercepted by the OS kernel. An area is used to spool the output so that a users' job does not have to wait for the printer to be available. One can examine a print queue status by using *lpq* and *lpstat* commands in Unix.

Clocks : The CPU has a system clock. The OS uses this clock to provide a variety of system- and application-based services. For instance, the print-out should display the date and time of printing. Below we list some of the common clock-based services.

- Maintaining time of day. (Look up date command under Unix.)
- Scheduling a program run at a specified time during systems' operation. (Look up *at* and *cron* commands under Unix.)
- Preventing overruns by processes in preemptive scheduling. Note that this is important for real-time systems. In RTOS one follows a scheduling policy like the earliest deadline first. This policy may necessitate preemption of a running process.
- Keeping track of resource utilization or reserving resource use.
- Performance related measurements (like timing IO, CPU activity).

Addressing a device: Most OSs reserve some addresses for use as exclusive addresses for devices. A system may have several DMA controllers, interrupt handling cards (for some process control), timers, serial ports (for terminals) or terminal concentrators, parallel ports (for printers), graphics controllers, or floppy and CD ROM drives, etc. A

fixed range of addresses allocated to each of these devices. This ensures that the device drives communicate with the right ports for data.

Caching: A cache is an intermediate level fast storage. Often caches can be regarded as fast buffers. These buffers may be used for communication between disk and memory or memory and CPU. The CPU memory caches may used for instructions or data. In case cache is used for instructions, then a group of instructions may be pre-fetched and kept there. This helps in overcoming the latency experienced in instruction fetch. In the same manner, when it is used for data it helps to attain a higher locality of reference.

As for the main memory to disk caches, one use is in disk rewrites. The technique is used almost always to collect all the write requests for a few seconds before actually a disk is written into. Caching is always used to enhance the performance of systems.

IO channels: An IO channel is primarily a small computer to basically handle IO from multiple sources. It ensures that IO traffic is smoothed out.

OS and CDE: The common desk top environment (CDE) is the norm now days. An OS provides some terminal-oriented facilities for operations in a CDE. In particular the graphics user interface (GUI) within windows is now a standard facility. The kernel IO system recognizes all cursor and mouse events within a window to allow a user to bring windows up, iconize, scroll, reverse video, or even change font and control display. The IO kernel provides all the screen management functions within the framework of a CDE.

5.6 Motivation For Disk Scheduling

Primary memory is volatile whereas secondary memory is non-volatile. When power is switched off the primary memory loses the stored information whereas the secondary memories retain the stored information. The most common secondary storage is a disk.

In Figure 5.13 we describe the mechanism of a disk and information storage on it. A disk has several platters. Each platter has several rings or tracks. The rings are divided into sectors where information is actually stored. The rings with similar position on different

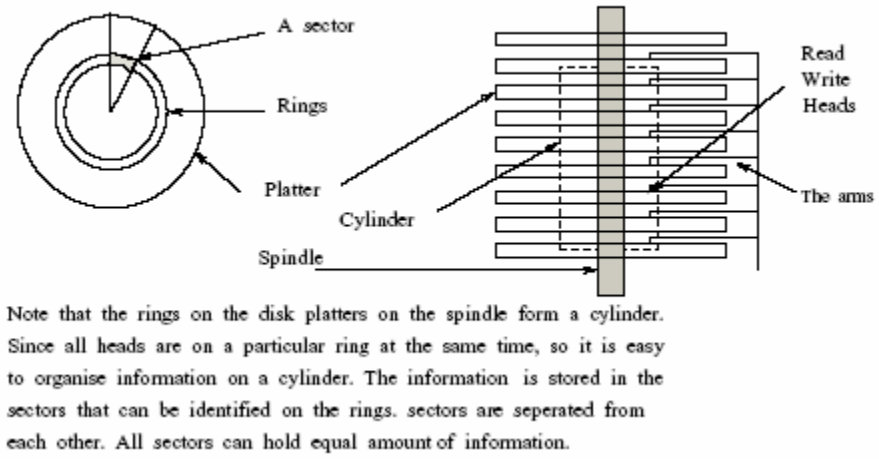

Figure 5.13: Information storage organisation on disks.

platters are said to form a cylinder. As the disk spins around a spindle, the heads transfer the information from the sectors along the rings. Note that information can be read from the cylinder surface without any additional lateral head movement. So it is always a good idea to organize all sequentially-related information along a cylinder. This is done by first putting it along a ring and then carrying on with it across to a different platter on the cylinder. This ensures that the information is stored on a ring above or below this ring. Information on different cylinders can be read by moving the arm by relocating the head laterally. This requires an additional arm movement resulting in some delay, often referred to as seek latency in response. Clearly, this delay is due to the mechanical structure of disk drives. In other words, there are two kinds of mechanical delays involved in data transfer from disks. The seek latency, as explained earlier, is due to the time required to move the arm to position the head along a ring. The other delay, called rotational latency, refers to the time spent in waiting for a sector in rotation to come under the read or write head. The seek delay can be considerably reduced by having a head per track disk. The motivation for disk scheduling comes from the need to keep both the delays to a minimum. Usually a sector which stores a block of information, additionally has a lot of other information. In Figure 5.14 we see that a 512 byte block has nearly 100 bytes of additional information which is utilized to synchronize and also check correctness of the information transfer as it takes place. Note that in figure 5.14 we have two pre-ambles each of 25 bytes, two synchronizing bytes, 6 bytes for checking errors in data transfer and a post-amble.

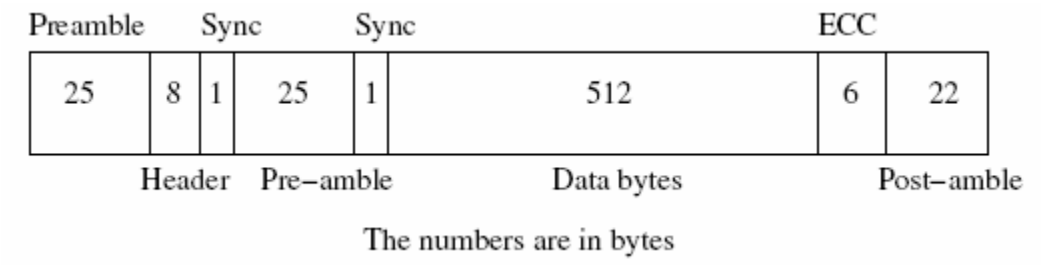

Figure 5.14: Information storage in sectors.

Scheduling Disk Operations: A user as well as a system spends a lot of time of operation communicating with files (programs, data, system utilities, etc.) stored on disks. All such communications have the following components:

1. The IO is to read from, or write into, a disk.
2. The starting address for communication in main memory
3. The amount of information to be communicated (in number of bytes / words)
4. The starting address in the disk and the current status of the transfer.

The disk IO is always in terms of blocks of data. So even if one word or byte is required we must bring in (or write in) a block of information from (to) the disk. Suppose we have only one process in a system with only one request to access data. In that case, a disk access request leads finally to the cylinder having that data. However, because processor and memory are much faster than disks, it is quite possible that there may be another request made for disk IO while the present request is being serviced. This request would queue up at the disk. With multi-programming, there will be many user jobs seeking disk access. These requests may be very frequent. In addition, the information for different users may be on completely different cylinders. When we have multiple requests pending on a disk, accessing the information in a certain order becomes very crucial. Some policies on ordering the requests may raise the throughput of the disk, and therefore, that of the system.

Let us consider an example in which we have some pending requests. We only need to identify the cylinders for these requests. Suppose, there are 200 tracks or rings on each platter. We may have pending requests that may have come in the order 59, 41, 172, 74, 52, 85, 139, 12, 194, and 87.

The FCFS policy: The first come first served policy entails that the service be provided strictly in the sequence in which the requests arrived. If we do that then we service in the sequence 59, 41, 172, 74, 52, 85, 139, 12, 194, and 87. It is a good practice to analyze the

effect of implementing a certain policy. In this case we try to analyze it by mapping the arm movements. The arm movement captures the basic disk activity. In the next Section we consider other policies as well. We also compare the arm movement required for FCFS policy with those required for the other policies.

5.7 Disk Scheduling Policies

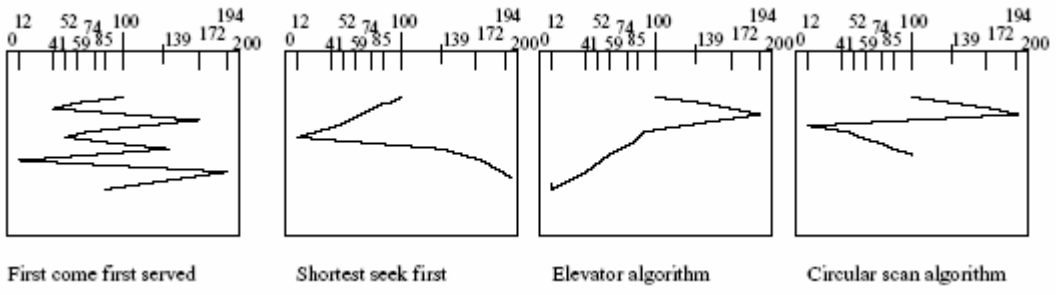

Figure 5.15: Comparison of disk scheduling policies.

In FCFS policy the information seek takes place in the sequence of cylinder numbers which are in the request queue. In figure 5.15 we plot map for FCFS access. In all the examples we assume that the arm is located at the cylinder number 100. The arm seek fluctuates sometimes very wildly. For instance, in our example there are wild swings from 12 to 194 and 41 to 172.

Shortest seek first: We look at the queue and compute the nearest cylinder location from the current location. Following this argument, we shall access in the order 87, 85, 4, 59, 52, 41, 12, 139, 172, and finally 194.

Elevator algorithm: Let us assume an initial arm movement in a certain direction. All the requests in that direction are serviced first. We have assumed that at 100 the arm as moving in the direction of increasing cylinder numbers. In that case it shall follow the sequence 139, 172, 194 and then descend to 87, 85, 74, 59, 41, and 12.

Circular scan: In the C-scan policy service is provided in one direction and then wraps round. In our example if the requests are serviced as the cylinder numbers increase then he sequence we follow would be 139, 172, and 174 and then wrap around to 12, 41, 52, 59, 74, 85, and finally 87.

From the response characteristics we can sense that FCFS is not a very good policy. In contrast, the shortest seek first and the elevator algorithm seems to perform well as these have the least arm movements. The circular scan too could be a very good scheduling

mechanism, if the fly-back time for the disk arm is very short. In this chapter we have explored IO mechanisms. The IO devices are also resources.

Besides the physical management of these resources, OS needs to have a strategy for logical management of the resources as well. In the next chapter we shall discuss resource management strategies.