I/O Management and Disk Scheduling

1 Diversity of I/O devices

There are all kinds of I/O devices in an operating system. Although they never play a centric role in the system, the management of these devices are absolutely the messiest aspect of operating system design.

Why? I/O devices may differ from one another on all kinds of aspects. Some are **human readable**, suitable for communicating with the computer users, such as printers, monitors, speakers, keyboards, and mice. Some are **machine readable**, suitable for communicating with electronic equipments, such as disks and tapes. Some others are otherwise used for **communicating with remote devices**, such as modems and network cards.

The above categories are obtained based on the functions of I/O devices. What's more, even those that fall into one category may still be different from one another in various aspects. They may have different data transferring speed, different complexity of control, etc.

2 Operating system design issues

Two objects are paramount in designing the I/O management facility: generality and efficiency.

2.1 Generality

We always desire to manage all the I/O devices in a uniform manner, however the above differences make it difficult to achieve this goal, both from the point of view of the operating system and from the point of view of user processes.

What we can do is to use a hierarchical approach to the design of the I/O function, which is similar to the abstraction scheme in object orientation theory. That is the routines that are involved in dealing with accessing I/O devices are divided into several layers, and each layer provides a concise interface for the immediate upper-level layer, so that most of the details

of the I/O devices are hidden in lower-level layers and the user processes may operate the devices in terms of general functions, such as *read*, *write*, *open*, *close*, *lock*, *unlock*.

What kind of hierarchy should be used depends on the type of I/O devices of concern. Figure 1 shows the organization of three different types of I/O devices.

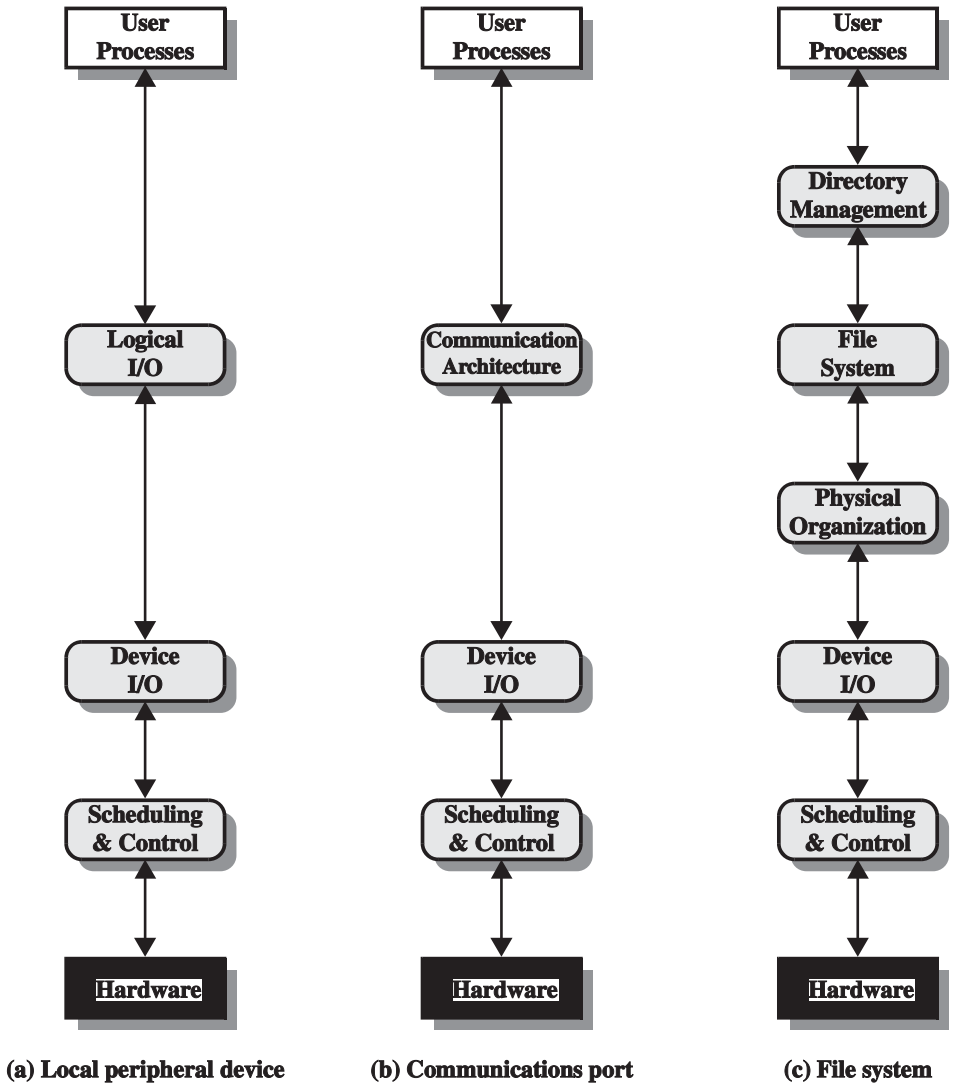

Figure 1: A model of I/O organization

Figure 1 (a) depicts the simplest case, a local peripheral device, which may be accessed in terms of a stream of bytes or records. The layers involved are as follows:

- **Logical I/O** deals with the I/O device as a logical resource without concerning the details of actually controlling the device. This layer provides a simple interface for user processes: specifying a unique ID to this device and simple control commands, such as *open*, *close*, *read*, *write*, etc..

- **Device I/O** converts the requested operation from the logical I/O layer into a series of I/O instructions. Data may be buffered in this layer and forwarded in the format that is acceptable to the destination layer.
- **Scheduling and control**: actually interacts with the I/O device, including queuing and scheduling of I/O operations, handling interrupts, collecting and reporting status information, etc.

For a communication device, rather than logical I/O, a communication protocol layer is used instead as Figure 1 (b) shows.

For a storage device on which a file system is used, more complexity is involved. Besides scheduling and control layer and device I/O layer, the following layers are also used:

- **Directory management** provides user processes a hierarchical view of the collection of files stored on the device. Operations may be performed to *add*, or *delete* a directory.
- **File system** provides a flat view of a collection of files. At this layer, a directory file is treated in the same way as other regular files, although the content of this special file includes information about files under this directory. The upper level may use this interface of this layer to *open*, *close*, *read*, *write*, *add*, or *delete* a file. Access control may also be included here.
- **Physical organization** converts the logical references to files into physical addresses in terms of track, sector. Allocation and deallocation of storage space are also treated in this layer.

2.2 Efficiency

Efficiency is the other objective in I/O subsystem design. Remember, at the beginning of this course, we have discussed three popular ways to access I/O devices:

- **Programmed I/O** means the microprocessor is involved all the way through the I/O process. The microprocessor issues command to initiate the I/O device, tell it what to do, and wait until the I/O process is finished. Almost all I/O devices are slower than the microprocessor, so the latter has to be idle for a long time, which results in inefficiency.
- **Interrupt-driven I/O** otherwise aims to let the microprocessor and the I/O devices work simultaneously. With this scheme, the microprocessor, after issuing a command on behalf of a process to start an I/O operation, may turn to execute another process. When

the I/O operation completes, an I/O interrupt will be generated to signal the microprocessor to resume the former process. Although switching between different processes involves overhead, it is still worth doing unless the context switching is too frequent.

- **DMA** When a large amount of data needs to be transferred to/from an I/O device, it is not realistic any longer to take the interrupt-driven approach because frequent context switching is unavoidable in this case. To deal with this problem, the DMA approach is taken, in which a DMA controller is responsible to control the data exchange between main memory and an I/O device. What the microprocessor needs to do is simply to send necessary information to the DMA controller at the very beginning. An interrupt is also generated finally so that the microprocessor knows the completion of the whole DMA process.

By considering whether the processor is directly involved in the I/O process and whether interrupt schemes are used or not, we may have Table 1, showing the similarity and difference among the above three schemes:

Table 1: I/O techniques

	No Interrupts	**Use of Interrupts**
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer	-	Direct memory access (DMA)

From the above discussion, we can see the efficiency of *the whole system* is increased step by step more or less.

3 I/O buffering

Let's begin with an example to introduce the necessity of I/O buffering. Suppose a user process wishes to read a block of data from a tape into a data area in the address space of this process at virtual location 1000. The most straightforward way to do so in the process is to execute an I/O command to the tape device and then wait for the data to be available.

However there are two problems with this approach. First, the process has to wait a long time before the completion of the operation due to the low speed of the tape device. Second, this data exchange may interfere with the swapping mechanism in the operation system. That is, before the whole operation is finished, the data area in the process to contain data from the

tape has to remain in the main memory. Thus this section of space must be locked in some way so that it will not be swapped out to secondary memory.

To avoid the above inefficiency and inconvenience, it is possible to perform data input transfers in advance of requests being made or output transfers after the request is made, which is known as *buffering*. This section will discuss some of buffering schemes to improve the performance of the system.

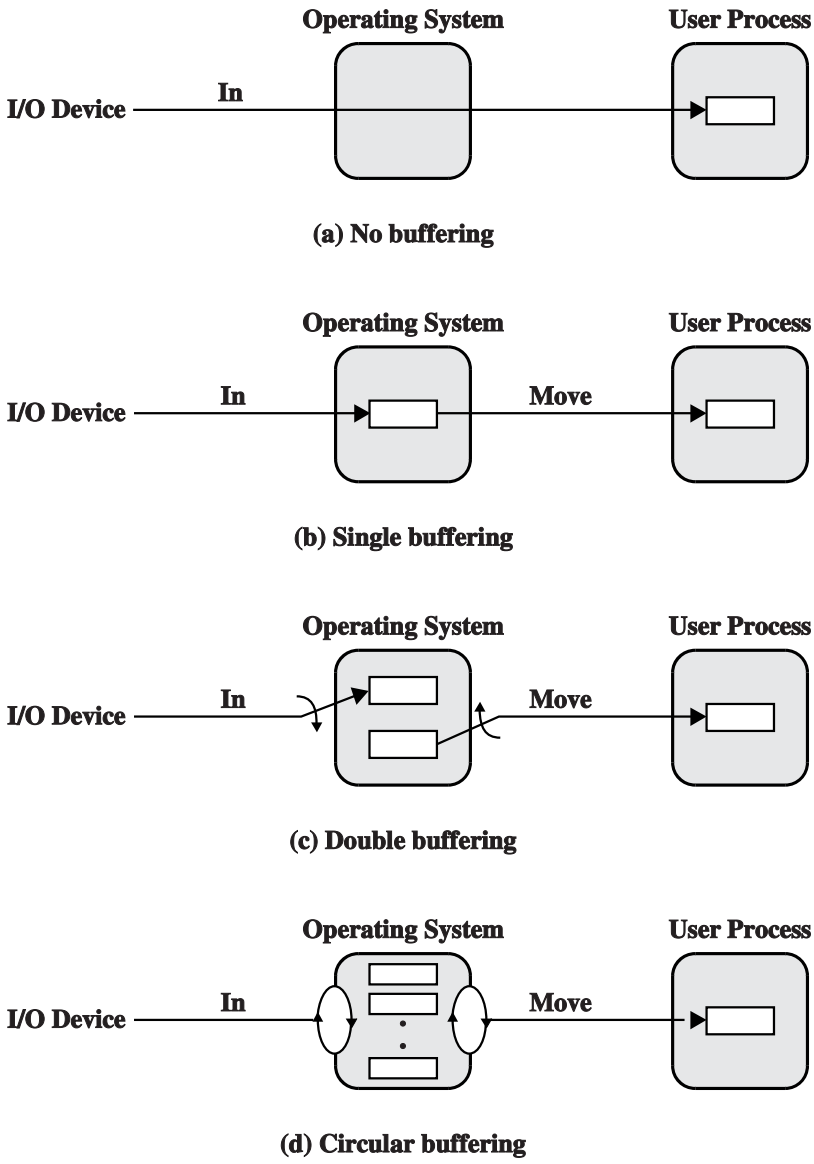

Figure 2: I/O buffering schemes

3.1 Single buffer

The simplest buffering scheme is *single buffering*. As illustrated in Figure 2 (b), when a user process issues an I/O request, the operating system allocates a buffer in the system portion of main memory to the operation.

With this scheme, data is first gradually transferred into the system buffer. When the transfer is complete, the data is then moved into the user process space. Note that this is not the end of the story. The I/O facility in the operating system continues to read more data from the device, which is called *reading ahead*. This is performed because user processes usually access data on an I/O device sequentially.

Thus this approach will generally speedup the I/O operations requested by a process. After a block of data becomes available and is delivered to the user process, the operation system may read ahead another block of data while the process is processing the former block. This approach also makes it possible to swap out the process that made I/O requests.

To support this mechanism, more effort needs to be made in the operating system, such as tracking the buffers allocated to user processes. What's more, the swapping of the process though is made possible, another problem arises. That is if the I/O request has been made and the process is to be swapped out to the same secondary storage device as involved in the I/O request, then the swapping will not happen until after the I/O request is finished. Nevertheless, at this moment, it may not be appropriate any more to swap out the process, since the data it has been waiting for are available and it may proceed to process them. Despite these disadvantages, it is still worth using the buffering scheme.

A similar circumstance will occur regarding outbound data transfer. When data are to be outputted to a device, they may be copied from the requesting process into a system buffer first. Then the process may simply continue or be swapped out.

3.2 Double buffer

An improvement of single buffering is to use two system buffers for a user process, called *double buffering*, as Figure 2 (c) shows. In this case, moving data from one system buffer to the process and reading data from the I/O device to the other buffer may be performed simultaneously. Again, this improvement comes at the cost of increased complexity.

3.3 Circular buffer

The idea of double buffering may be generalized to *circular buffering*. That is more than two buffers are used and the collection of these buffers is referred to as a circular buffer, as illus-

trated in Figure 2 (d). In this way, although the I/O device is much slower, there may be enough data in the system buffers for the process to read.

4 Disk Scheduling

Above we discussed how I/O access is improved step by step from programmed I/O to DMA and how I/O buffering could be used, but the focus there is the I/O subsystem in general. We may also improve the I/O performance based on the characteristics of various concrete I/O devices. Disks are the most commonly used I/O devices in a computer. This section talks about how to schedule I/O accesses to disks.

4.1 Disk performance parameters

A hard disk is usually made up of multiple platters, as illustrated in Figure 3, each of which use two heads to write and read data, one for the top of the platter and one for the bottom (this isn't always the case, but usually is). Either side of each platter is made up of multiple tracks, which in turn are divided into several sectors, as depicted in Figure 4.

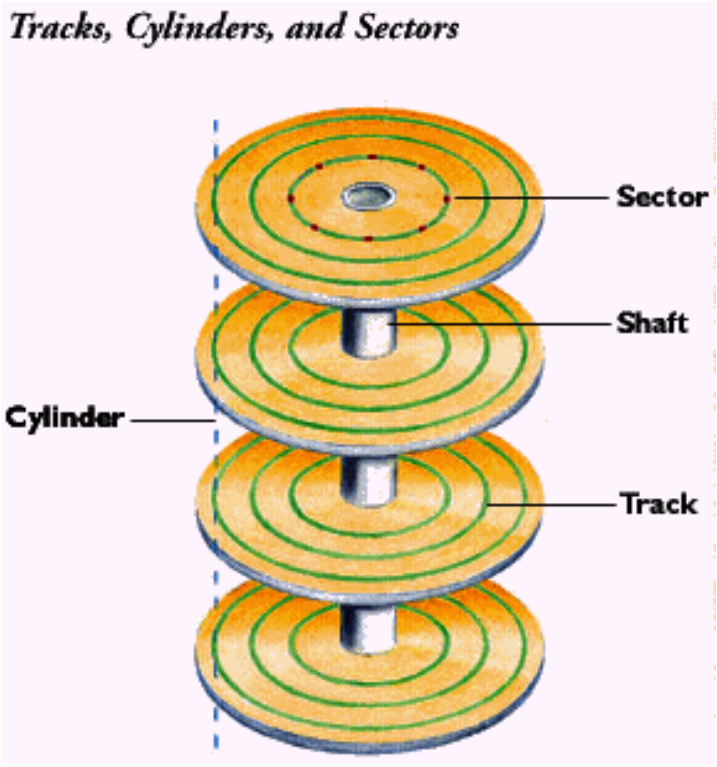

Figure 3: The typical structure of a hard disk

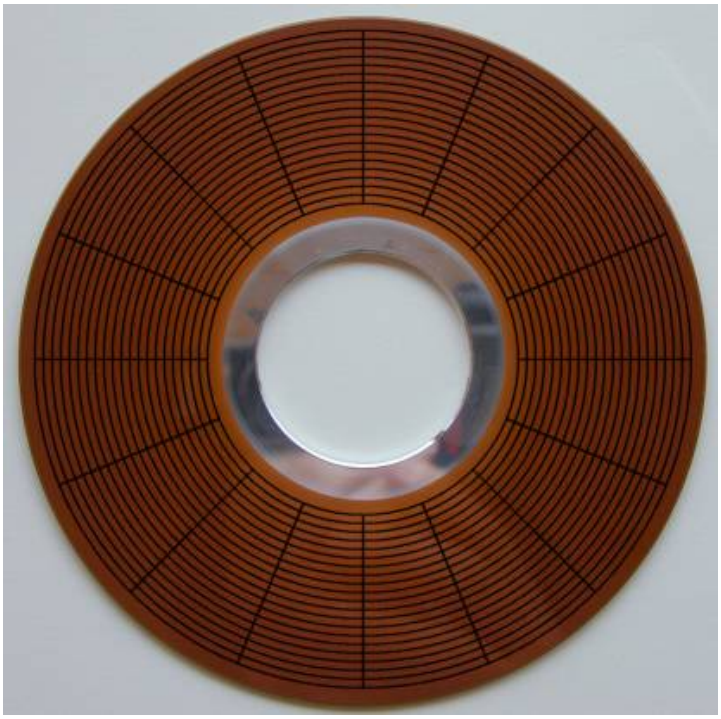

Figure 4: The layout of one side of a platter

The heads that access the platters are locked together on an assembly of head arms, which may move in only two directions, towards the spindle or the opposite. This means that all the heads move in and out together, so each head is always physically located at the same track number. It is not possible to have one head at track 0 and another at track 1,000. Besides the movement of the heads, the spindle may rotate so that the heads may access a specific sector on the track they are located at.

And because of this arrangement, often the track location of the heads is not referred to as a track number but rather as a cylinder number. A cylinder is basically the set of all tracks that all the heads are currently located at. The addressing of individual sectors of the disk is traditionally done by referring to cylinders, heads and sectors (CHS).

When the disk drive is operating, the disk is rotating at constant speed. Thus once the head is positioned at the desired track, it simply waits until the desired sector is under it. The time taken for the head to move is known as **seek time** and the time taken for the desired sector to become available to the head is known as **rotational delay** or **rotational latency**. The sum of these two delays is the **access time**. Once the head is in position, the read or write operation may then be performed, which is the data transfer portion of the operation. Besides these two kinds of time cost, an I/O request may probably also have to wait in a queue of I/O requests until the corresponding device becomes available. Based on the analysis, the general timing diagram of disk I/O transfer is shown in Figure 5.

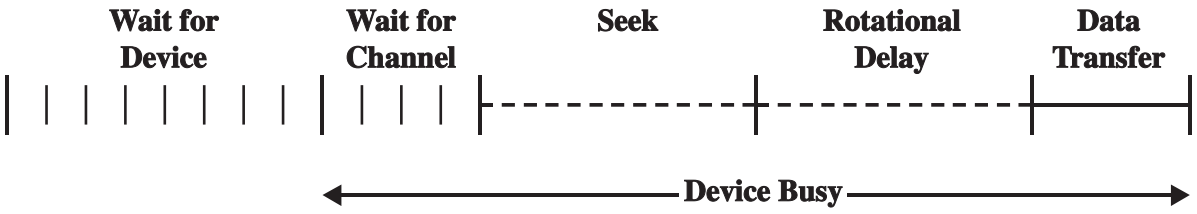

Figure 5: Timing of a disk I/O transfer

If we use T_s for seek time, T_r for rotational delay, and T for transfer time, then the total average access time can be expressed as:

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

where r is rotation speed of the disk in revolutions per second, b is the number of bytes to be transferred, and N is the number of bytes on a track.

Obviously T_s and T_r for each specific I/O request are totally determined by the physical disk device and there is no way to improve the performance in operating system design by reducing these values. However things are different if we consider multiple I/O requests as a whole and schedule them elegantly. For example, there are 3 requests, respectively for a sector at Track 4, 5 and 9. Obviously Track 4 and 5 should be accessed consecutively before or after Track 9 is accessed. We should always avoid the access order of Track 4, 9, and 5. The factor that makes difference here is seek time.

4.2 Disk scheduling policies

We have shown by an example that disk I/O performance may be improved by scheduling multiple requests in the purpose of minimizing the sum of seek times of these I/O requests.

Typically, the operating system maintains a queue of request for each I/O device. Thus for each disk, we may consider all kinds of scheduling algorithms and compare their performances. We may select **random scheduling** as a benchmark to evaluate other strategies.

Suppose we have a disk with 200 tracks and the requested tracks, in the order received, are 55, 58, 39, 18, 90, 160, 150, 38, 184. Figure 6 shows the comparison of different disk request scheduling algorithms.

4.2.1 FIFO

FIFO is again the most straightforward policy to schedule the requests, in which the requests are processed in the same order as they are received. This strategy has the advantage of being

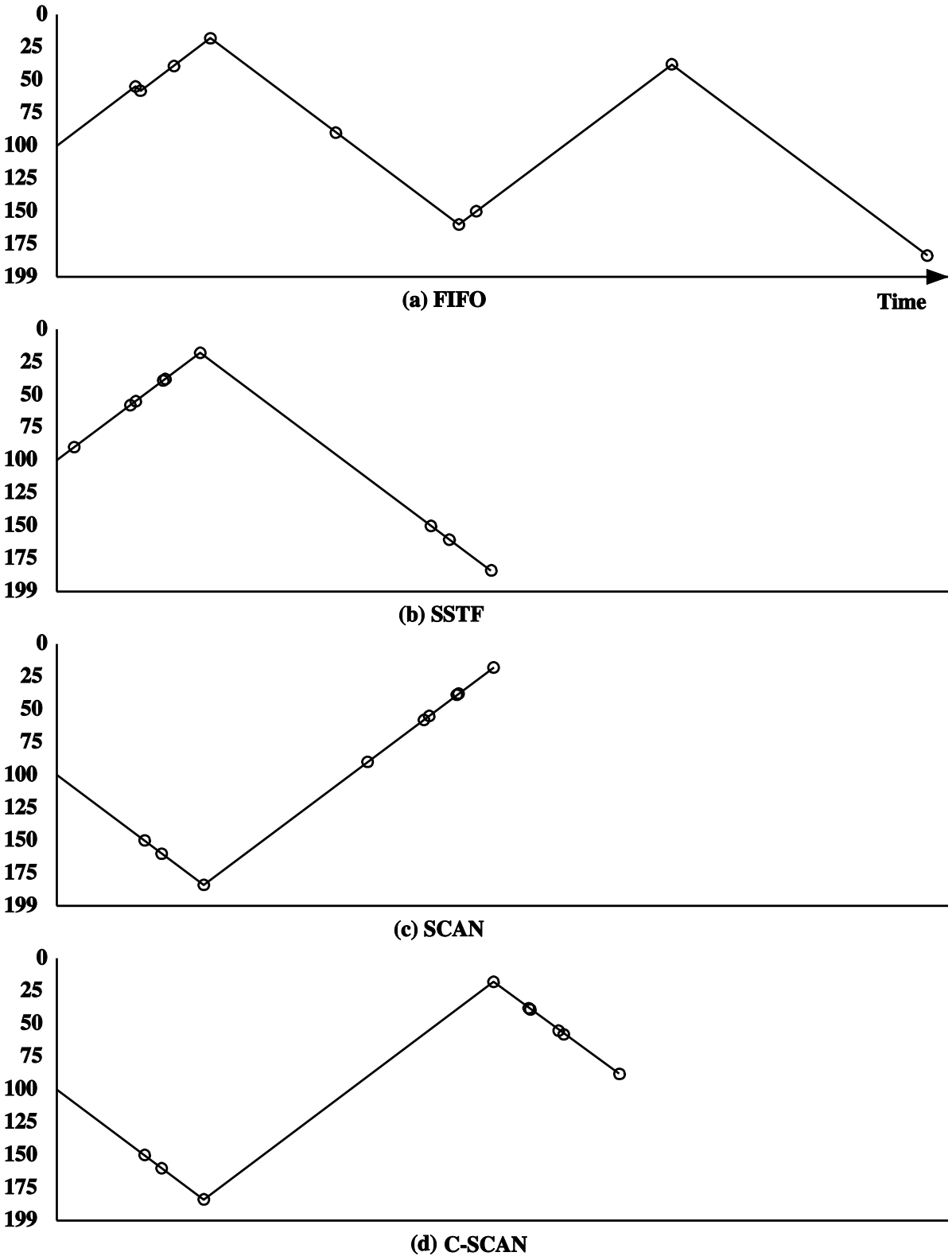

Figure 6: Comparison of disk scheduling algorithms

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	**Average seek length**	27.5	**Average seek length**	27.8	**Average seek length**	35.8

Table 2: Comparison of disk scheduling algorithms

fair, however it behaves to much extent like random scheduling because I/O requests come in in a random manner and are not likely to access tracks in the way described by the principle of locality.

4.2.2 Priority-Based

Again as in other types of scheduling, different priorities may be assigned to different requests, for example requests made by short processes or interactive processes may be assigned higher priorities, which helps to provide good interactive response time. However longer processes may have to wait excessively long times or even suffer from starvation. Obviously FIFO does not take advantage of the location information of tracks requested.

4.2.3 Shortest Service Time First

The SSTF policy is to select the disk I/O request that requires the least movement of the disk arm from its current position. Although a series of optimal decisions each at one step do not guarantee an overall optimal solution, this policy should have a better performance than FIFO. Figure 6 (b) and Table 2 (b) show the performance of SSTF on our example assuming the initial location of the read/write head is Track 100.

4.2.4 SCAN

The SSTF policy clearly may lead to the starvation of some requests if new requests come in constantly and are scheduled before the former. A simple alternative that prevents this kind

of starvation is the SCAN policy, which mimics the behavior of an elevator.

With SCAN, the arm keeps moving in one direction, fulfilling all outstanding requests en route, until it reaches the last track in that direction or there are no more requests ahead. The latter refinement is sometimes referred to as the LOOK policy. Figure 6 (c) and Table 2 (c) show the performance of SCAN, which is almost the same as the SSTF policy, though in theory, SCAN has a better performance than SSTF.

It is not difficult to see that the SCAN policy favors the tracks in the middle over the innermost and outer-most ones since within one cycle of header movement, the latter are reaches only once but all the others twice.

4.2.5 C-SCAN

To avoid the problem in SCAN, a circular SCAN policy (C-SCAN) may be used. With C-SCAN, when the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again. In this way, each track receives identical service.

5 Disk cache

Finally to speed up the disk I/O access, a disk cache, which is actually a portion of main memory, may be used. The principle of the use of a disk cache is all the same as the cache between a micro-processor and the main memory. Some replacement algorithms are also needed when a miss event occurs. Please refer to the text for more information if more interest.