[redacted]

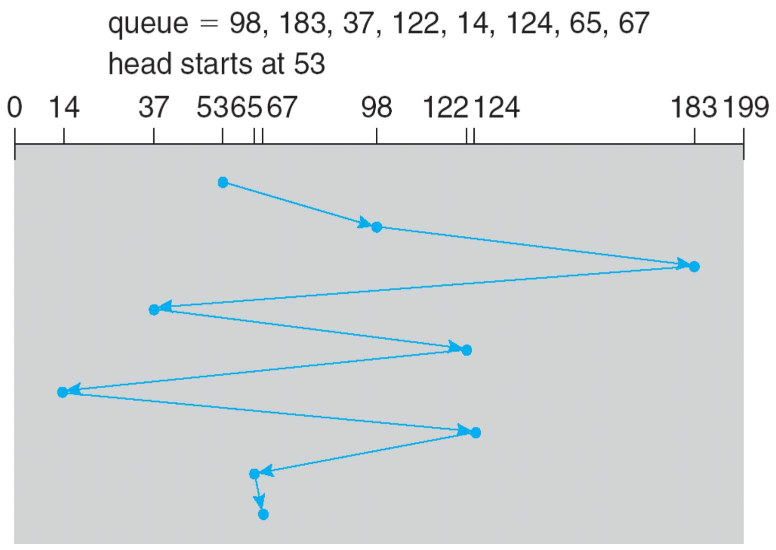
HDD OPERATION

We begin with an example request queue for a Hard Disk Drive (HDD) containing requests for sectors 98, 183, 37, 122, 14, 124, 65, and 67. In order to design an efficient HDD system, we must ask ourselves: What do we wish to optimize? Since any computer system must consider long-term performance, we choose to optimize throughput over latency. This means that, on average, we will service a higher number of requests per unit time. However, the average wait time for any one request may increase.

First, it the use of a buffer in HDD requests warrants discussion. Why should we service a group of requests instead of a single request? Would it not be faster to immediately service requests as they arrive? It turns out that is not the case. There is a significant performance gain to be achieved by implementing a buffer. We can often increase overall throughput (which is our goal) when we are allowed to optimize a sequence of requests. There are various algorithms possible to approach that optimization challenge. We will discuss a few of them. First, we look at the most simple: First Come First Serve (FCFS).

<u>FCFS:</u>

FCFS management of a HDD is not ideal. This graphic shows the head movement using FCFS for our request buffer above:



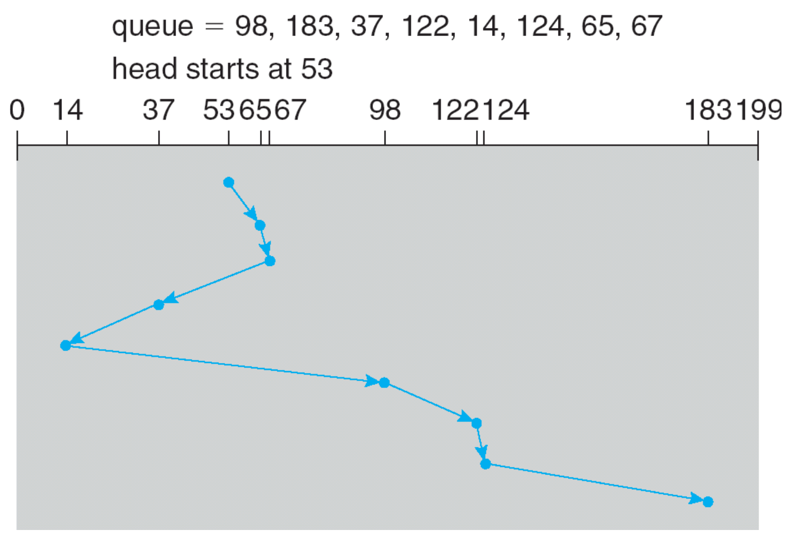
As you can see, the movement is sporadic and un-ordered. Rather than servicing a request near the current head-location (which would be very fast), we move the head to a far-away location to serve the next request in the buffer.

[redacted]

In fact, any benefit to utilizing a buffer is lost under this algorithm. Since we are simply using the buffer as a FIFO queue, we may as well service requests directly.

Shortest Seek Time First (SSTF):

This algorithm takes into account the current head position and considers all requests in the buffer. The request with memory location nearest to the current head position is chosen to be serviced. This is shown here:



The result is minimal seek time, which then leads to maximum throughput. However, as always, there must be a downside.

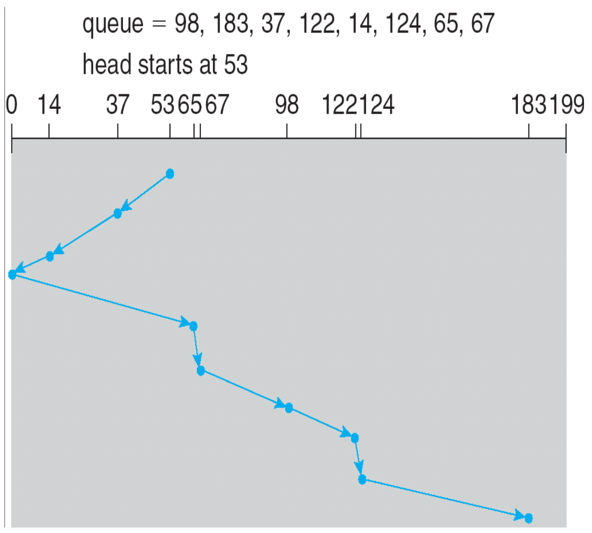
It was suggested that a large enough buffer could lead to long wait times under this algorithm. While this is true, real-world HDD systems actually implement very small buffers for that reason. So, that is not the downfall we are looking for.

The downfall, it turns out, is potential starvation. If a request far from the current head position arrives, but all subsequent requests are near the current head position, our first request will be forced to wait (potentially forever). This results in starvation of that request, and that is obviously not desirable.

SCAN (The Elevator Algorithm):

The SCAN algorithm causes a HDD head to function like an elevator. Under this algorithm, we not only consider current head location, but also current head direction. So, if the head is at sector x and moving down, it will not change direction until it has reached the lowest possible sector. Along the way, the head will stop and service any requests that fall in its path. This is illustrated here:

[redacted]

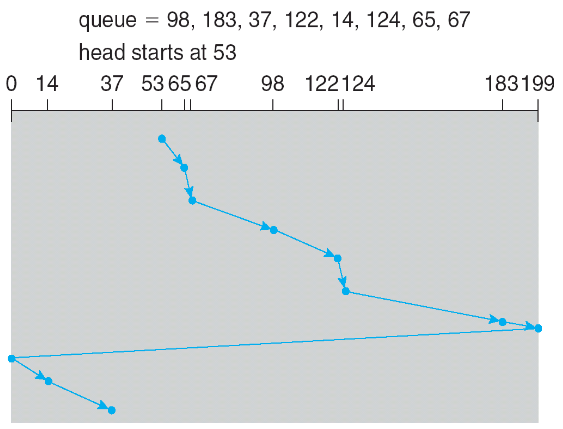


Similarly, an elevator going down will not pass by a passenger who is also waiting to go down without servicing that passenger. And, an elevator will not change direction until it has reached the highest or lowest request.

This algorithm sacrifices some of the performance gain from SSTF, but adds some level of fairness. Since the head is constantly scanning from edge to edge, every request is guaranteed to not starve.

C-SCAN:

Like the SCAN algorithm, C-SCAN constantly moves the head from one extreme to the other. However, under C-SCAN, requests are only serviced when the head is moving in one of the two possible directions. For example, the head may begin at the inner extreme and move inwards while servicing requests. Once it reaches the inner extreme, however, the head will move as quickly as possible back to the outer edge without servicing any requests along the way. The process would then repeat. This is shown here:



[redacted]

<u>C-LOOK:</u>

We will not discuss this algorithm in detail. However, it is important to note that other algorithms do exist. These other algorithms not only consider track seek time, but also rotational seek time. This adds a new layer of complexity that we will not dive into.

Next, we discussed the concept of <u>Anticipatory Scheduling</u>. First, we consider the concept of work conservation. That is to say, if there is a request in the buffer, the disk head will move. This is not always ideal when considering HDD systems when we take into account the concept of spatial locality. Spatial locality says that nearby sectors are likely to be accessed around the same time. So, if we service a request for address x and then move to a far away address for the next request, we have a relatively high risk of receiving a new request for somewhere around x (due to spatial locality). Anticipatory scheduling is the answer to this new concern.

One implementation that is often used for anticipatory scheduling is a slight delay before allowing the disk head to move to a far-away track. This type of scheduling algorithm is generally implemented in the OS or filesystem software, however.

[redacted]

OS MANAGEMENT OF I/O DEVICES

As we already know, the HDD is significantly slower than CPU/memory. Because of this, the OS must use various techniques to manage the delay associated with disk access. The first technique we discussed is the use of buffers.

Consider an fwrite() function call in a user program. If the function call were to pass its data directly to the destination disk (write-through), then the program would experience a significant delay in waiting for the function to return. So, in order to reduce that delay, the OS uses a buffer. Instead of the function call writing directly to the disk, its data is put into a buffer (write-back). The function is then allowed to return. From there, the buffer serves various purposes:

- Contiguous (or nearby) write requests are grouped together. This allows the disk the easily optimize the seek sequence and reduce latency.
- The buffer is dumped at some frequency by the OS. This results in a few large requests to the disk instead of many small requests. Due to the mechanical nature of the disk, this is ideal.
- If a read request comes along soon after a corresponding write request, then it is likely that the data still resides in the buffer. Since the buffer is much faster than the HHD, the data is read directly from the buffer. This results in faster reads.
    - For this reason, file system buffers are often referred to as Buffer-Caches.

So, as you can see, buffers provide various benefits. There is, however, a potential downfall. Since buffers are stored in a volatile medium, the data is not very secure. Should there be a power failure, data that was in the buffer waiting to be written to disk is lost.

It is also worth noting the existence of hardware efforts to overcome HDD slowness. One such example is RAID systems. RAID arrays consist of multiple HDDs linked together. Data is then stored in parallel (and sometimes redundantly) in order to increase throughput and reliability.

We now turn to discussion of the file system. The main goal of a file system is to provide performance and reliability when working with the HDD while maintaining the abstraction of files.

Reliability is achieved by adding extra information to stored data. This information can consist of redundancy and/or error correcting information. Strategies used here are very similar to those used in networking.

[redacted]

Performance is largely dictated by allocation method. The discussion of allocation method in regard to a HDD is similar to memory management, but with some key differences. Here, the question is which sectors should be allocated to a file.

The first, and perhaps simplest, allocation method used in HDDs is Contiguous Allocation. Whereas contiguous allocation in memory was a poor choice, the same is not necessarily true here. As a review, contiguous allocation will allocate a set of continuous sectors to a file. This prevents internal fragmentation, but leads to external fragmentation. In the context of a HDD, however, external fragmentation is not a critical concern. Since HDD space is cheap and abundant, we do not have the same conservation concerns we had with memory. Fragmentation is still not ideal, but it is tolerable here.

Contiguous allocation has two major benefits:

- The amount of information stored in the File Control Block (a structure maintained in the OS heap similar to a PCB or TCB) is minimal and independent of file size.
- Contiguous allocation leads to faster access time when reading from a file. Once we position the head at the beginning of the file, all subsequent requests (for that file) are in nearby sectors. So, the seek time is minimal.

This is where we ended our discussion for the day. Next class, we will continue with other allocation methods.