36

Hard Disk Drives

The last chapter introduced the general concept of an I/O device and showed you how the OS might interact with such a beast. In this chapter, we dive into more detail about one device in particular: the **hard disk drive**. These drives have been the main form of persistent data storage in computer systems for decades and much of the development of file system technology (coming soon) is predicated on their behavior. Thus, it is worth understanding the details of a disk's operation before building the file system software that manages it. Many of these details are available in excellent papers by Ruemmler et al. [RW92] and Anderson et al. [ADR03].

CRUX: HOW TO STORE AND ACCESS DATA ON DISK
How do modern hard-disk drives store data? What is the interface? How is the data actually laid out and accessed? How does disk scheduling work?

36.1 The Interface

Let's start by understanding the interface to a modern disk drive. The basic interface for all modern drives is straightforward. The drive consists of a large number of sectors (512-byte blocks), each of which can be read or written. The sectors are numbered from 0 to $n-1$ on a disk with n sectors. Thus, we can view the disk as an array of sectors; 0 to $n-1$ is thus the **address space** of the drive.

Multi-sector operations are possible; indeed, many file systems will read or write 4KB at a time (or more). However, when updating the disk, the only guarantee drive manufactures make is that a single 512-byte write is **atomic** (i.e., it will either complete in its entirety or it won't complete at all); thus, if an untimely power loss occurs, only a portion of a larger write may complete (sometimes called a **torn write**).

There are some assumptions most clients of disk drives make, but that are not specified directly in the interface; Schlosser and Ganger have called this the "unwritten contract" of disk drives [SG04]. Specifically, one can usually assume that accessing two blocks that are near one-another within the drive's address space will be faster than accessing two blocks that are far apart. One can also usually assume that accessing blocks in a contiguous chunk (i.e., a sequential read or write) is the fastest access mode, and usually much faster than any more random access pattern.

36.2 Basic Geometry

Let's start to understand some of the components of a modern disk. We start with a **platter**, a circular hard surface on which data is stored persistently by inducing magnetic changes to it. A disk may have one or more platters; each platter has 2 sides, each of which is called a **surface**. These platters are usually made of some hard material (such as aluminum), and then coated with a thin magnetic layer that enables the drive to persistently store bits even when the drive is powered off.

The platters are all bound together around the **spindle**, which is connected to a motor that spins the platters around (while the drive is powered on) at a constant (fixed) rate. The rate of rotation is often measured in **rotations per minute (RPM)**, and typical modern values are in the 7,200 RPM to 15,000 RPM range. Note that we will often be interested in the time of a single rotation, e.g., a drive that rotates at 10,000 RPM means that a single rotation takes about 6 milliseconds (6 ms).

Data is encoded on each surface in concentric circles of sectors; we call one such concentric circle a **track**. A single surface contains many thousands and thousands of tracks, tightly packed together, with hundreds of tracks fitting into the width of a human hair.

To read and write from the surface, we need a mechanism that allows us to either sense (i.e., read) the magnetic patterns on the disk or to induce a change in (i.e., write) them. This process of reading and writing is accomplished by the **disk head**; there is one such head per surface of the drive. The disk head is attached to a single **disk arm**, which moves across the surface to position the head over the desired track.

36.3 A Simple Disk Drive

Let us now understand how this all works by building up our model of how a disk drive works, one track at a time. Assume we have a very simple disk, with only a single track (Figure 36.1):

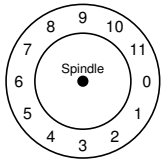

Figure 36.1: A Disk With Just A Single Track

This track has just 12 sectors, each of which is 512 bytes in size (our typical sector size, recall) and addressed therefore by the numbers 0 through 11. The single platter we have here rotates around the spindle, to which a motor is attached. Of course, the track by itself isn't too interesting; we want to be able to read or write those sectors, and thus we need a disk head, attached to a disk arm, as we now see (Figure 36.2).

In the figure, the disk head, attached to the end of the arm, is positioned over sector 6, and the surface is rotating in a counterclockwise fashion.

Single-track Latency: The Rotational Delay

To understand how a request would be processed on our simple, one-track disk, imagine we now receive a request to read block 0. How should the disk service this request?

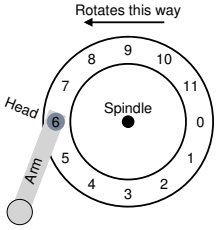

Figure 36.2: A Single Track Plus A Head

In our simple disk, the disk doesn't have to do much. In particular, it must just wait for the desired sector to rotate under the disk head. This wait happens often enough in modern drives, and is an important enough component of I/O service time, that it has a special name: **rotational delay** (sometimes **rotation delay**, though doesn't that sound weird?). In the example, if the full rotational delay is R, the disk would have to incur a rotational delay of about $\frac{R}{2}$ to wait for 0 to come under the read/write head (if we start at 6). A worst-case request on this single track would be to sector 5, causing nearly a full rotational delay in order to service such a request.

Multiple Tracks: Seek Time

So far our disk just has a single track, which is not too realistic; modern disks of course have many thousands of tracks. Let's thus look at ever-so-slightly more realistic disk surface, this one with three tracks (Figure 36.3).

In the figure, the head is currently positioned over the innermost track (which contains sectors 24 through 35); the next track over contains the next set of sectors (12 through 23), and the outermost track contains the first sectors (0 through 11).

To understand how the drive might access a given sector, we now trace what would happen on a request to a distant sector, e.g., a read to sector 11. To accomplish this read, the drive has to first move the disk arm to the correct track (in this case, the outermost track), in a process known as a **seek**. Seeks, along with rotations, are one of the most costly disk operations.

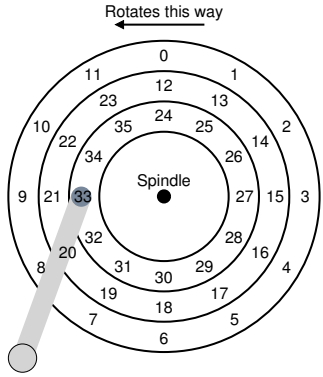

Figure 36.3: Three Tracks Plus A Head

The seek, it should be noted, has many phases: first an *acceleration* phase as the disk arm gets moving; then *coasting* as the arm is moving at full speed, then *deceleration* as the arm slows down; finally *settling* as the head is carefully positioned over the correct track. The **settling time** is often quite significant, e.g., 0.5 to 2 ms, as the drive must be certain to find the right track (imagine if it just got close instead!).

After the seek, the disk arm has positioned the head over the right track. Thus, the state of the disk might look like this (Figure 36.4).

As we can see in the diagram, during the seek, the arm has been moved to the desired track, and the platter of course has rotated, in this case about 3 sectors. Thus, sector 9 is just about to pass under the disk head, and we must only endure a short rotational delay to complete the transfer.

When sector 11 passes under the disk head, the final phase of I/O will take place, which is known as the **transfer**, where data is either read from or written to the surface. And thus, we have a complete picture of I/O time: first a seek, then waiting for the rotational delay, and finally the transfer.

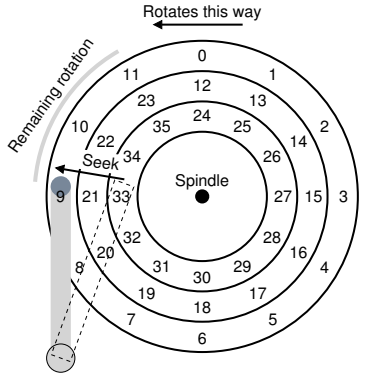

Figure 36.4: Three Tracks Plus A Head: After Seeking

Some Other Details

Though we won't spend too much time on it, there are some other interesting details about how hard drives operate. Many drives employ some kind of **track skew** to make sure that sequential reads can be properly serviced even when crossing track boundaries. In our simple example disk, this might appear as seen in Figure 36.5.

Sectors are often skewed like this because when switching from one track to another, the disk needs time to reposition the head (even to neighboring tracks). Without such skew, the head would be moved to the next track but the desired next block would have already rotated under the head, and thus the drive would have to wait almost the entire rotational delay to access the next block.

Another reality is that outer tracks tend to have more sectors than inner tracks, which is a result of geometry; there is simply more room out there. These tracks are often referred to as **multi-zoned** disk drives, where the disk is organized into multiple zones, and where a zone is consecutive set of tracks on a surface. Each zone has the same number of sectors per track, and outer zones have more sectors than inner zones.

Finally, an important part of any modern disk drive is its **cache**, for historical reasons sometimes called a **track buffer**. This cache is just some small amount of memory (usually around 8 or 16 MB)

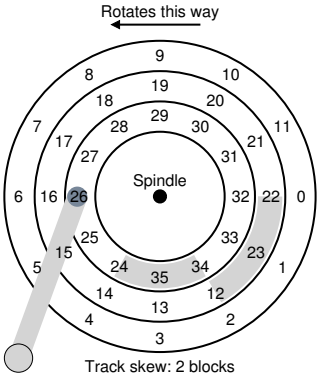

Figure 36.5: Three Tracks: Track Skew Of 2

which the drive can use to hold data read from or written to the disk. For example, when reading a sector from the disk, the drive might decide to read in all of the sectors on that track and cache them in its memory; doing so allows the drive to quickly respond to any subsequent requests to the same track.

On writes, the drive has a choice: should it acknowledge the write has completed when it has put the data in its memory, or after the write has actually been written to disk? The former is called **write back** caching (or sometimes **immediate reporting**), and the latter **write through**. Write back caching sometimes makes the drive appear "faster", but can be dangerous; if the file system or applications require that data be written to disk in a certain order for correctness, write-back caching can lead to problems (e.g., read about journaling).

36.4 I/O Time: Doing The Math

Now that we have an abstract model of the disk, we can use a little analysis to better understand disk performance. In particular, we can now represent I/O time as the sum of the three major components of I/O time:

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer} \tag{36.1}$$

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

Table 36.1: Disk Drive Specs: SCSI Versus SATA

Note that the rate of I/O ($R_{I/O}$), which is often more easily used for comparison between drives (as we will do below), is easily computed from the time. Simply divide the size of the transfer by the time it took:

$$R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}} \tag{36.2}$$

To get a better feel for I/O time, let us perform the following calculation. Assume there are two workloads we are interested in. The first, known as the **random** workload, issues small (e.g., 4KB) reads to random locations on the disk. Random workloads are common in many important applications, including database management systems. The second, known as the **sequential** workload, simply reads a large number of sectors consecutively from the disk, without jumping around. Sequential access patterns are quite common and thus important as well.

To understand the difference in performance between random and sequential workloads, we need to make a few assumptions about the disk drive first. Let's look at a couple of modern disks from Seagate. The first, known as the Cheetah 15K.5 [S09b], is a high-performance SCSI drive. Its performance characteristics are found in Table 36.1. The second, the Barracuda [S09a], is a drive built for capacity; its characteristics are also found in the table.

As you can see, the drives have quite different characteristics, and in many ways nicely summarize two important components of the disk drive market. The first is the "high performance" drive market, where drives are engineered to spin as fast as possible, deliver low seek times, and transfer data quickly. The second is the "capacity"

	Cheetah	Barracuda
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$R_{I/O}$ Sequential	125 MB/s	105 MB/s

Table 36.2: Disk Drive Performance: SCSI Versus SATA

market, where cost per byte is the most important aspect; thus, the drives are slower but pack as many bits as possible into the space available.

From these numbers, we can start to calculate how well the drives would do under our two workloads outlined above. Let's start by looking at the random workload. Assuming each 4 KB read occurs at a random location on disk, we can calculate how long each such read would take. On the Cheetah:

$$T_{seek} = 4\ ms,\ T_{rotation} = 2\ ms,\ T_{transfer} = 30\ microsecs \quad (36.3)$$

The average seek time (4 milliseconds) is just taken as the average time reported by the manufacturer; note that a full seek (from one end of the surface to the other) would likely take two or three times longer. The average rotational delay is calculated from the RPM directly. 15000 RPM is equal to 250 RPS (rotations per second); thus, each rotation takes 4 ms. On average, the disk will encounter a half rotation and thus 2 ms is the average time. Finally, the transfer time is just the size of the transfer over the peak transfer rate; here it is vanishingly small (30 *microseconds*; note that we need 1000 microseconds just to get 1 millisecond!).

Thus, from our equation above, $T_{I/O}$ for the Cheetah roughly equals 6 ms. To compute the rate of I/O, we just divide the size of the transfer by the average time, and thus arrive at $R_{I/O}$ for the Cheetah under the random workload of about 0.66 MB/s. The same calculation for the Barracuda yields a $T_{I/O}$ of about 13.2 ms, more than twice as slow, and thus a rate of about 0.31 MB/s.

Now let's look at the sequential workload. Here we can assume there is a single seek and rotation before a very long transfer. For simplicity, assume the size of the transfer is 100 MB. Thus, $T_{I/O}$ for the Barracuda and Cheetah is about 800 ms and 950 ms, respectively. The rates of I/O are thus very nearly the peak transfer rates of 125 MB/s and 105 MB/s, respectively. Table 36.2 summarizes these numbers.

ASIDE: **COMPUTING THE "AVERAGE" SEEK**

In many books and papers, you will see average disk-seek time cited as being roughly one-third of the full seek time. Where does this come from?

Turns out it arises from a simple calculation based on average seek *distance*, not time. Imagine the disk as a set of tracks, from 0 to N. The seek distance between any two tracks x and y is thus computed as the absolute value of the difference between them: $|x - y|$.

To compute the average seek distance, all you need to do is to first add up all possible seek distances:

$$\sum_{x=0}^{N}\sum_{y=0}^{N}|x-y|. \tag{36.4}$$

Then, divide this by the number of different possible seeks: N^2. To compute the sum, we'll just use the integral form:

$$\int_{x=0}^{N}\int_{y=0}^{N}|x-y|\,\mathrm{d}y\,\mathrm{d}x. \tag{36.5}$$

To compute the inner integral, let's break out the absolute value:

$$\int_{y=0}^{x}(x-y)\,\mathrm{d}y+\int_{y=x}^{N}(y-x)\,\mathrm{d}y. \tag{36.6}$$

Solving this leads to $(xy-\frac{1}{2}y^2)\Big|_0^x+(\frac{1}{2}y^2-xy)\Big|_x^N$ which can be simplified to $(x^2-Nx+\frac{1}{2}N^2)$. Now we have to compute the outer integral:

$$\int_{x=0}^{N}(x^2-Nx+\frac{1}{2}N^2)\,\mathrm{d}x, \tag{36.7}$$

which results in:

$$(\frac{1}{3}x^3-\frac{N}{2}x^2+\frac{N^2}{2}x)\Big|_0^N \;=\; \frac{N^3}{3}. \tag{36.8}$$

Remember that we still have to divide by the total number of seeks (N^2) to compute the average seek distance: $(\frac{N^3}{3})/(N^2) \;=\; \frac{1}{3}N$. Thus the average seek distance on a disk, over all possible seeks, is one-third the full distance. And now when you hear that an average seek is one-third of a full seek, you'll know where it came from.

TIP: USE DISKS SEQUENTIALLY
When at all possible, transfer data to and from disks in a sequential manner. If I/O is done in little random pieces, I/O performance will suffer dramatically. Also, users will suffer. Also, you will suffer, knowing what suffering you have wrought with your random I/Os.

The table shows us a number of important things. First, and most importantly, there is a huge gap in drive performance between random and sequential workloads, almost a factor of 200 or so for the Cheetah and more than a factor 300 difference for the Barracuda. And thus we arrive at the most obvious design tip in the history of computing.

A second, more subtle point: there is a large difference in performance between high-end "performance" drives and low-end "capacity" drives. For this reason (and others), people are often willing to pay top dollar for the former while trying to get the latter as cheaply as possible.

36.5 Disk Scheduling

Because of the high cost of I/O, the OS has historically played a role in deciding the order of I/Os issued to the disk. More specifically, given a set of I/O requests, the **disk scheduler** examines the requests and decides which one to schedule next.

Unlike job scheduling, where the length of each job is usually unknown, with disk scheduling, we can make a good guess at how long a "job" (i.e., disk request) will take. By estimating the seek and possible the rotational delay of a request, the disk scheduler can know how long each request will take, and thus (greedily) pick the one that will take the least time to service first. Thus, the disk scheduler will try to follow the **principle of SJF (shortest job first)** in its operation. For more details on disk scheduling, see either [SCO90] or [JW91], which both provide good overviews.

SSTF: Shortest Seek Time First

One early disk scheduling approach is known as **shortest-seek-time-first** (**SSTF**) (also called **shortest-seek-first** or **SSF**). SSTF orders the

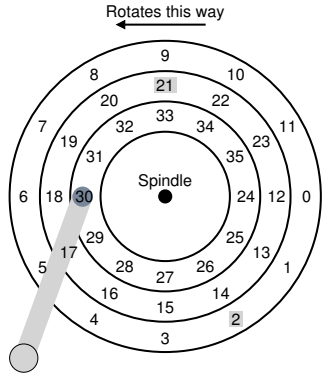

Figure 36.6: SSTF: Scheduling Requests 21 And 26

queue of I/O requests by track, and picks the request on the nearest track to complete first. For example, assuming the current position of the head is over the inner track, and we have requests for sectors 21 (middle track) and 2 (outer track), we would then issue the request to 21 first, wait for it to complete, and then issue the request to 2.

SSTF works well in this example, seeking to the middle track first and then the outer track. However, there are some problems with SSTF that this example does not make clear. First, as we discussed above, the drive geometry is not available to the host OS; rather, it sees an array of blocks. Fortunately, this problem is rather easily fixed. Instead of SSTF, an OS can simply implement **nearest-block-first** (**NBF**), which schedules the request with the nearest block address to the last request next.

The second problem is more fundamental: **starvation**. Imagine in our example above if there were a steady stream of requests to the inner track, where the head currently is positioned. Requests to any other tracks would then be ignored completely by a pure SSTF approach. And thus the crux of the problem:

CRUX: HOW TO HANDLE DISK STARVATION
How can we implement a SSTF-like scheduling algorithm but avoid starvation?

Elevator (a.k.a. SCAN or C-SCAN)

The answer to this query was developed some time ago (see [CKR72] for example), and is relatively straightforward. The algorithm, originally called **SCAN**, simply moves across the disk servicing requests in order across the tracks. Let us call a single pass across the disk a *sweep*. Thus, if a request comes for a block on a track that has already been serviced on this sweep of the disk, it is not handled immediately, but rather queued until the next sweep.

SCAN has a number of variants, all of which do about the same thing. For example, Coffman et al. introduced **F-SCAN**, which freezes the queue to be serviced when it is doing a sweep [CKR72]; this action places requests that come in during the sweep into a queue to be serviced later. Doing so avoids starvation of far-away requests, by delaying the servicing of late-arriving (but nearer by) requests.

C-SCAN is another common variant, short for **Circular SCAN**. Instead of sweeping in one direction across the disk, the algorithm sweeps from outer-to-inner, and then inner-to-outer, etc.

For reasons that should now be obvious, this algorithm (and its variants) is sometimes referred to as the **elevator** algorithm, because it behaves like an elevator which is either going up or down and not just servicing requests to floors based on which floor is closer. Imagine how annoying it would be if you were going down from floor 10 to 1, and somebody got on at 3 and pressed 4, and the elevator went up to 4 because it was "closer" than 1! As you can see, the elevator algorithm, when used in real life, prevents fights from taking place on elevators. In disks, it just prevents starvation.

Unfortunately, SCAN and its cousins do not represent the best scheduling technology. In particular, SCAN (or SSTF even) do not actually adhere as closely to the principle of SJF as they could. In particular, they ignore rotation. And thus, another crux:

CRUX: HOW TO ACCOUNT FOR DISK ROTATION COSTS
How can we implement an algorithm that more closely approximates SJF by taking *both* seek and rotation into account?

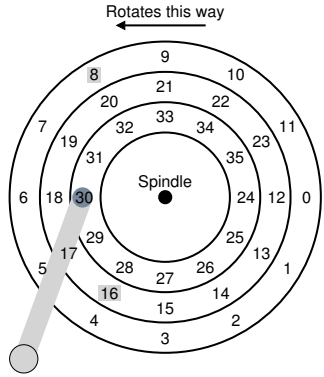

Figure 36.7: SSTF: Sometimes Not Good Enough

SPTF: Shortest Positioning Time First

Before discussing **shortest positioning time first** or **SPTF** scheduling (sometimes also called **shortest access time first** or **SATF**), which is the solution to our problem, let us make sure we understand the problem in more detail. Figure 36.7 presents an example.

In the example, the head is currently positioned over sector 30 on the inner track. The scheduler thus has to decide: should it schedule sector 16 (on the middle track) or sector 8 (on the outer track) for its next request. So which should it service next?

The answer, of course, is "it depends". In engineering, it turns out this is almost always the answer, so if you don't know an answer, you might want to go with it. However, it is almost always better to know *why* it depends.

TIP: IT ALWAYS DEPENDS (LIVNY'S LAW)
Almost any question can be answered with "it depends", as our colleague Miron Livny always says (and that's why we call it **Livny's Law**). However, use with caution, as if you answer too many questions this way, people will stop asking you questions altogether. For example, somebody asks: "want to go to lunch?" You reply: "it depends, are *you* coming along?"

What it depends on here is the relative time of seeking as compared to rotation. If, in our example, seek time is much higher than rotational delay, then SSTF (and variants) are just fine. However, imagine if seek is quite a bit faster than rotation. Then, in our example, it would make more sense to seek *further* to service request 8 on the outer track than it would to perform the shorter seek to the middle track to service 16, which has to rotate all the way around before passing under the disk head.

On modern drives, as we saw above, both seek and rotation are roughly equivalent (depending, of course, on the exact requests), and thus SPTF is useful. However, it is even more difficult to implement in an OS, which generally does not have a good idea where track boundaries are or where the disk head currently is (in a rotational sense).

Modern Scheduling Issues

One final issue we'd like to discuss before ending this note is how disk scheduling is done on modern systems. Older systems assumed the OS did all the scheduling, and the OS would only issue a single request at a time.

In modern systems, disks can accommodate multiple outstanding requests, and have sophisticated internal schedulers themselves (which can implement SPTF accurately, for example, as in the disk you know all of the relevant details). Thus, the OS scheduler usually picks what it thinks the best few requests are and issues them to disk; the disk then uses its internal knowledge of head position and detailed track layout to service said requests in the best possible (SPTF) order.

36.6 Summary

We have presented a summary of how disks work. The summary is actually a detailed functional model; it does not describe the amazing physics, electronics, and material science that goes into actual drive design. For those interested in even more details of that nature, we suggest a different major (or perhaps minor); for those that are happy with this model, good! We can now proceed to using the model to build more interesting systems on top of these incredible devices.

References

[ADR03] "More Than an Interface: SCSI vs. ATA"
Dave Anderson, Jim Dykes, Erik Riedel
FAST '03, 2003
One of the best recent-ish references on how modern disk drives really work; a must read for anyone interested in knowing more.

[CKR72] "Analysis of Scanning Policies for Reducing Disk Seek Times"
E.G. Coffman, L.A. Klimko, B. Ryan
SIAM Journal of Computing, September 1972, Vol 1. No 3.
Some of the early work in the field of disk scheduling.

[JW91] "Disk Scheduling Algorithms Based On Rotational Position"
D. Jacobson, J. Wilkes
Technical Report HPL-CSP-91-7rev1, Hewlett-Packard (February 1991)
A more modern take on disk scheduling. It remains a technical report (and not a published paper) because the authors were scooped by Seltzer et al. [SCO90].

[RW92] "An Introduction to Disk Drive Modeling"
C. Ruemmler, J. Wilkes
IEEE Computer, 27:3, pp. 17-28, March 1994
A terrific introduction to the basics of disk operation. Some pieces are out of date, but most of the basics remain.

[SCO90] "Disk Scheduling Revisited"
Margo Seltzer, Peter Chen, John Ousterhout
USENIX 1990
A paper that talks about how rotation matters too in the world of disk scheduling.

[SG04] "MEMS-based storage devices and standard disk interfaces: A square peg in a round hole?"
Steven W. Schlosser, Gregory R. Ganger
FAST '04, pp. 87-100, 2004
While the MEMS aspect of this paper hasn't yet made an impact, the discussion of the contract between file systems and disks is wonderful and a lasting contribution.

[S09a] "Barracuda ES.2 data sheet"
`http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah_15k_5.pdf`
A data sheet; read at your own risk. Risk of what? Boredom.

[S09b] "Cheetah 15K.5"
`http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_es.pdf`
See above commentary on data sheets.

Homework

This homework uses `disk.py` to familiarize you with how a modern hard drive works. It has a lot of different options, and unlike most of the other simulations, has a graphical animator to show you exactly what happens when the disk is in action.

Let's do a simple example first. To run the simulator and compute some basic seek, rotation, and transfer times, you first have to give a list of requests to the simulator. This can either be done by specifying the exact requests, or by having the simulator generate some randomly.

We'll start by specifying a list of requests ourselves. Let's do a single request first:

```
prompt> disk.py -a 10
```

At this point you'll see:

```
...
REQUESTS ['10']

For the requests above, compute the seek, rotate, and transfer times.
Use -c or the graphical mode (-G) to see the answers.
```

To be able to compute the seek, rotation, and transfer times for this request, you'll have to know a little more information about the layout of sectors, the starting position of the disk head, and so forth. To see much of this information, run the simulator in graphical mode (-G):

```
prompt> disk.py -a 10 -G
```

At this point, a window should appear with our simple disk on it. The disk head is positioned on the outside track, halfway through sector 6. As you can see, sector 10 (our example sector) is on the same track, about a third of the way around. The direction of rotation is counter-clockwise. To run the simulation, press the "s" key while the simulator window is highlighted.

When the simulation completes, you should be able to see that the disk spent 105 time units in rotation and 30 in transfer in order to access sector 10, with no seek time. Press "q" to close the simulator window.

To calculate this (instead of just running the simulation), you would need to know a few details about the disk. First, the rotational speed is by default set to 1 degree per time unit. Thus, to make a complete revolution, it takes 360 time units. Second, transfer begins and ends at the halfway point between sectors. Thus, to read sector 10, the transfer begins halfway between 9 and 10, and ends halfway between 10 and 11. Finally, in the default disk, there are 12 sectors per track, meaning that each sector takes up 30 degrees of the rotational space. Thus, to read a sector, it takes 30 time units (given our default speed of rotation).

With this information in hand, you now should be able to compute the seek, rotation, and transfer times for accessing sector 10. Because the head starts on the same track as 10, there is no seek time. Because the disk rotates at 1 degree / time unit, it takes 105 time units to get to the beginning of sector 10, halfway between 9 and 10 (note that it is exactly 90 degrees to the middle of sector 9, and another 15 to the halfway point). Finally, to transfer the sector takes 30 time units.

Now let's do a slightly more complex example:

```
prompt> disk.py -a 10,11 -G
```

In this case, we're transferring two sectors, 10 and 11. How long will it take? Try guessing before running the simulation!

As you probably guessed, this simulation takes just 30 time units longer, to transfer the next sector 11. Thus, the seek and rotate times remain the same, but the transfer time for the requests is doubled. You can in fact see these sums across the top of the simulator window; they also get printed out to the console as follows:

```
...
Sector:  10  Seek:  0  Rotate:105  Transfer: 30  Total: 135
Sector:  11  Seek:  0  Rotate:  0  Transfer: 30  Total:  30
TOTALS      Seek:  0  Rotate:105  Transfer: 60  Total: 165
```

Now let's do an example with a seek. Try the following set of requests:

```
prompt> disk.py -a 10,18 -G
```

To compute how long this will take, you need to know how long a seek will take. The distance between each track is by default 40

distance units, and the default rate of seeking is 1 distance unit per unit time. Thus, a seek from the outer track to the middle track takes 40 time units.

You'd also have to know the scheduling policy. The default is FIFO, though, so for now you can just compute the request times assuming the processing order matches the list specified via the `-a` flag.

To compute how long it will take the disk to service these requests, we first compute how long it takes to access sector 10, which we know from above to be 135 time units (105 rotating, 30 transferring). Once this request is complete, the disk begins to seek to the middle track where sector 18 lies, taking 40 time units. Then the disk rotates to sector 18, and transfers it for 30 time units, thus completing the simulation. But how long does this final rotation take?

To compute the rotational delay for 18, first figure out how long the disk would take to rotate from the end of the access to sector 10 to the beginning of the access to sector 18, assuming a zero-cost seek. As you can see from the simulator, sector 10 on the outer track is lined up with sector 22 on the middle track, and there are 7 sectors separating 22 from 18 (23, 12, 13, 14, 15, 16, and 17, as the disk spins counter-clockwise). Rotating through 7 sectors takes 210 time units (30 per sector). However, the first part of this rotation is actually spent seeking to the middle track, for 40 time units. Thus, the actual rotational delay for accessing sector 18 is 210 minus 40, or 170 time units. Run the simulator to see this for yourself; note that you can run without graphics and with the `-c` flag to just see the results without seeing the graphics.

```
prompt> ./disk.py -a 10,18 -c
...
Sector:  10  Seek:  0  Rotate:105  Transfer: 30  Total: 135
Sector:  18  Seek: 40  Rotate:170  Transfer: 30  Total: 240
TOTALS      Seek: 40  Rotate:275  Transfer: 60  Total: 375
```

You should now have a basic idea of how the simulator works. The questions below will explore some of the different options, to better help you build a model of how a disk really works.

Questions

1. Compute the seek, rotation, and transfer times for the following sets of requests: `-a 0`, `-a 6`, `-a 30`, `-a 7,30,8`, and finally `-a 10,11,12,13`.

2. Do the same requests above, but change the seek rate to different values: `-S 2`, `-S 4`, `-S 8`, `-S 10`, `-S 40`, `-S 0.1`. How do the times change?

3. Do the same requests above, but change the rotation rate: `-R 0.1`, `-R 0.5`, `-R 0.01`. How do the times change?

4. You might have noticed that some request streams would be better served with a policy better than FIFO. For example, with the request stream `-a 7,30,8`, what order should the requests be processed in? Now run the shortest seek-time first (SSTF) scheduler (`-p SSTF`) on the same workload; how long should it take (seek, rotation, transfer) for each request to be served?

5. Now do the same thing, but using the shortest access-time first (SATF) scheduler (`-p SATF`). Does it make any difference for the set of requests as specified by `-a 7,30,8`? Find a set of requests where SATF does noticeably better than SSTF; what are the conditions for a noticeable difference to arise?

6. You might have noticed that the request stream `-a 10,11,12,13` wasn't particularly well handled by the disk. Why is that? Can you introduce a track skew to address this problem (`-o skew`, where `skew` is a non-negative integer)? Given the default seek rate, what should the skew be to minimize the total time for this set of requests? What about for different seek rates (e.g., `-S 2`, `-S 4`)? In general, could you write a formula to figure out the skew, given the seek rate and sector layout information?

7. Multi-zone disks pack more sectors into the outer tracks. To configure this disk in such a way, run with the `-z` flag. Specifically, try running some requests against a disk run with `-z 10,20,30` (the numbers specify the angular space occupied by a sector, per track; in this example, the outer track will be packed with a sector every 10 degrees, the middle track every

20 degrees, and the inner track with a sector every 30 degrees). Run some random requests (e.g., `-a -1 -A 5,-1,0`, which specifies that random requests should be used via the `-a -1` flag and that five requests ranging from 0 to the max be generated), and see if you can compute the seek, rotation, and transfer times. Use different random seeds (`-s 1`, `-s 2`, etc.). What is the bandwidth (in sectors per unit time) on the outer, middle, and inner tracks?

8. Scheduling windows determine how many sector requests a disk can examine at once in order to determine which sector to serve next. Generate some random workloads of a lot of requests (e.g., `-A 1000,-1,0`, with different seeds perhaps) and see how long the SATF scheduler takes when the scheduling window is changed from 1 up to the number of requests (e.g., `-w 1` up to `-w 1000`, and some values in between). How big of scheduling window is needed to approach the best possible performance? Make a graph and see. Hint: use the `-c` flag and don't turn on graphics with `-G` to run these more quickly. When the scheduling window is set to 1, does it matter which policy you are using?

9. Avoiding starvation is important in a scheduler. Can you think of a series of requests such that a particular sector is delayed for a very long time given a policy such as SATF? Given that sequence, how does it perform if you use a **bounded SATF** or **BSATF** scheduling approach? In this approach, you specify the scheduling window (e.g., `-w 4`) as well as the BSATF policy (`-p BSATF`); the scheduler then will only move onto the next window of requests when *all* of the requests in the current window have been serviced. Does this solve the starvation problem? How does it perform, as compared to SATF? In general, how should a disk make this trade-off between performance and starvation avoidance?

10. All the scheduling policies we have looked at thus far are **greedy**, in that they simply pick the next best option instead of looking for the optimal schedule over a set of requests. Can you find a set of requests in which this greedy approach is not optimal?