

# 1 I/O in general

I/O devices are highly varied. A computer display, a hard drive, and a mouse are all considered I/O devices. An OS has to support communicating with and controlling specific I/O devices (this is what a device driver actually is). This device specific software has to be exposed to the programmer through a series of layers that abstract it to some idealized logical device that the user/programmer can use in a generalized (i.e. not device-specific) way.

## 1.1 Simple Example: The Mouse

First, consider the lowly mouse. A computer mouse doesn't have to be particularly fast. Compared to people, computers are blindingly fast, so mouse inputs occur once in a blue moon as far as the CPU is concerned. However, the mouse generates unpredictable inputs at random intervals, and so is a good candidate for interrupt-driven I/O. The inputs themselves are fairly simple too: just an x coordinate, a y coordinate, and indications of clicking. So, when the user uses the mouse, the mouse fires an interrupt. The OS processes the interrupt, which may involve:

1. Stopping the current process
2. Swapping in the OS's context
3. Reading in the mouse's inputs (x,y,click)
4. Writing them into the target process' memory
5. Informing the target process of a mouse click (by setting a flag, sending a signal, etc.)

Steps 3 and 4 can be finessed by using what's known as memory-mapped I/O. In memory-mapped I/O, I/O device control registers are mapped to specific memory addresses. In this case, the OS can either just let the user process read I/O addresses (which may be unsafe), or just needs to copy from the I/O addresses to the process' address space (which is faster and simpler than using I/O specific instructions).

## 2 Hard Drives: Our spinning, high capacity friends

*Appendix 11A (end of chapter 11)*

Hard drives are omnipresent and highly useful I/O devices. But what actually is a hard drive? A hard drive is a magnetic storage device, where bits are encoded in the alignments of magnetic domains. In modern drives, the domains exist in a material thinly coated on the surface of spinning discs (this is why they're also called hard disks). These discs are called platters, and in most hard drives, there's a stack of them all attached to the same axle (called a spindle). Data is read off of the platters by devices known as read/write heads. The heads hover just off the surface of the platters and can be moved in towards the spindle or out towards the edge. Each head is basically a tiny electro-magnet with a correspondingly tiny coil. To write, a current is sent through the coil, which

induces a magnetic field, which polarizes the bit of the platter just beneath the head. As the platter passes beneath the head, the magnetization of the surface will induce a tiny current in the head, which can be used to read out values. Each platter will have its own read/write head, and all the heads are connected together to form something called a comb. Because the comb (and therefore the heads) can only move back and forth, the platters have to spin beneath the heads so that the entire surface of the platter is available for reading and writing. (In fact the heads hover so close to the platters and the platters spin so fast that the situation is analogous to a 747 flying at 65 mph just 1.5mm off the ground).

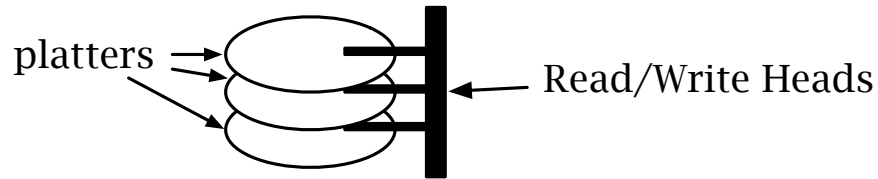


Figure 1: A Simplified View of a Hard Drive

## 2.1 Sectors, tracks, etc.

A platter is a circle with a hole in it, so it makes sense to talk about locations on a platter surface in terms of a series of concentric circles centered about the spindle. Each of the circles is called a track, and the back-and-forth movement of the read/write heads is used to move the head to another track. When there are multiple platters, tracks at the same distance from the spindle on each platter are grouped into a 'cylinder'. This makes sense, because all the read/write heads are connected, they all read the same track (on different platters) at the same time. Each track is subdivided into chunks of bits called sectors. Currently, most hard drives pack 512 bytes per sector, although it can range from 32 bytes up to 32K. The number of sectors per track can vary too, from 4-32 sectors per track. Usually the number of sectors is uniform across all tracks. This introduces an interesting problem, namely that if sector size is constant and the number of sectors per track is constant, that means that the bit density is lower at the edge of a platter than in the middle. Some disks spread out the sectors on the outer tracks, some disks keep the sectors the same size and just add more inter-sector padding, and some disks slowly increase the number of sectors per track as you move out from the spindle. However, for the most part, you can usually assume the same number of sectors per track across the platters.

In main memory, the fundamental addressable unit is usually the byte. On the hard drive, however, it's the sector. Because sectors are organized in tracks, to get to a specific sector one would also need to specify the track number. On a disk with multiple platters, one may also need to specify the platter. So the address of a sector on a hard drive would be a triplet: (*platter*, *track*, *sector*). The reason why most multi-platter drives deal with cylinders is that you can eliminate the platter specifier and just label sectors in order from top to bottom (or bottom to top). So on a system with 2

platters and 8 sectors per track, sector 0, track 0 on platter 1 would actually be sector 9 on cylinder 0. If the number of sectors per track is a power of two, then the platter specifier can actually be encoded in the top bits of the sector number (this is why the number of sectors is usually a power of two). In modern systems the primary elements of HD addressing are the sector and track.

## 2.2 Latency

Hard drives are slow (at least compared to main memory). How slow? Well, main memory can usually index a given address within  $\approx 10\text{ns}$  (a nano second is  $10^{-9}$  seconds). While a modern hard drive can index a given sector on the order of  $10\text{ms}$  (a millisecond is  $10^{-3}$  seconds). Here we see roughly a million-fold speed difference! That's a big deal. However, we still use harddrives because they are so much cheaper per bit than main memory. How much cheaper? Earlier this week I browsed around on the web and found that you can get an 80GB drive for \$38, while you can get 256M of RAM for \$4. So you'd need 320 sticks of 256M ram to match the storage, and that would set you back about \$1280. Of course, you might be able to get some kind of bulk-order deal and knock that down to \$640, which is still almost 17 times more expensive per bit than hard drive storage. Plus, when's the last time you saw a motherboard with space for 320 sticks of RAM? Additionally, this is super-cheap RAM (read: low-end). More advanced stuff costs even more per bit, an 8GB stick costs \$135, which is about 10% more per bit than the cheapo RAM. So hard drives are slow, but they're way cheaper than faster memory, so we use them anyway. But why are hard drives slow? When the read/write head is over the bits in question, the hard drive can read them fairly fast (on the order of  $20\ \mu\text{s}$  for 1K). But what if the head is at the right track, but the sector you want to read isn't under the head? Then, the drive has to wait for the sector to rotate back around so that the head can read off the bits. This waiting time is called "rotational latency" and is, on average, 4 - 5 ms for a 7200 RPM drive. This latency has to be measured as an average because a sector can be any fraction of the track away from the read/write head. So to get a meaningful number, you treat the location of a sector as a random variable distributed uniformly over the track. This is why server people care about the RPM rating of their hard drives. A 10000 RPM drive spins faster than a 7200 RPM drive (but of course it costs more). Now, what happens when the read/write head isn't even over the track you need? Well, then the entire comb has to be repositioned to the correct track (and then at that track, you may have to wait for the sector of interest to roll around). This is known as "seek time" and seems to be currently hovering around 9ms (on average). This also has to be measured as an average, for similar reasons to rotational latency. However, it's even worse for seek times. Most hard drives have fancy motors to drive the comb back and forth and the read/write heads are moved at faster speeds if you are seeking far across the drive rather than just a few tracks over. So, unlike rotational latency, the speed at which the heads move isn't constant! In general though, it's faster to seek to nearby tracks and slower to seek to far away tracks. Hard drives are slow, because they don't have a uniform access speed and their addressing latency is VERY high. For a hard drive recieving completely random sector and track requests, the chances that the read/write head is already right on top of the correct sector is very low (1 in 32768, for a drive with 1024 tracks and 32 sectors per track). So, generally speaking, the hard drive is going to have to move the read/write head (and pay the seek time cost) and then wait for the sector you want to roll around (and pay the rotational latency cost). What this

means is that the total time between when you ask for a sector and when it becomes available for the processor to read can range between 5 and 15 ms (13 on average). In 5ms, my laptop is capable of executing 23 MILLION instructions! Already it should be apparent that an important part of the OS's job is to reduce hard drive latency as much as possible (more on this in the scheduling section).

## 3 Drive Controls

Controlling a hard drive can be a complex thing. In the old days, a hard drive would just sit on the bus and would have to be controlled directly by the CPU. Modern systems are somewhat more sophisticated where the hard drives themselves are actually attached to a controller, which is basically a teeny special purpose processor. The CPU then asks the controller to grab data and the controller takes care of a lot of the grungy details. In fact, modern hard drives do all kinds of things internally that OSes use to have to do 20 years ago (e.g. bad block detection).

### 3.1 Direct Control

In a direct control scenario, all the hard drive can do is seek to a particular track, raise an interrupt when it's done seeking, and stream track data out over the bus. Let's say the OS need to read sector 3 from track 8:

1. CPU sends HD (SEEK track 8)
2. HD starts seeking, CPU resumes normal execution
3. HD finishes seeking, raises interrupt
4. CPU processes interrupt and starts reading bytes streaming off of HD onto the bus
5. CPU filters out the particular sector it was interested in, and stores it in main memory.

In this case, the CPU itself has to examine the so-called block stream coming from the hard drive. Data is divided into blocks (sized to conveniently fit into a sector), where a block is a collection of the data as well as meta-data about it. Some of this meta-data consists of organizational info (cylinder/track/sector#), sizing information, and synchronization. Additionally, error-detecting information may be present. This was particularly important back when OSes had to compensate for failing hard drives (bad blocks were the nightmare of many a DOS user). When you format your hard drive, you're basically writing out empty blocks to the disk.

#### 3.1.1 Checksum and Errors

Errors on a storage medium are something your average programmer doesn't have to deal with. So it can be a little odd to think about. Basically an error is when one or more bits have changed on disk independently of the OS. So, how would you detect an error? In general it's infeasible to

store another copy of the data around just for checking, so you need something more light weight. The usual tactic is to construct an algorithm that summarizes the data in the block, and that's fairly cheap to compute. An instance of this is the checksum. Basically you compute the checksum of all the bytes in a block in a single pass and store it in the block's meta-data (you do this each time you write the block to the disk). Next time you read the block off disk, you recompute the checksum and see if it matches the one on the disk (if not, you've just detected an error). A checksum is an error detection mechanism that uses mathematical properties of a sum of bits in order to help determine when an error has occurred. In general, you add up all the bytes, drop the carry, and take the two's complement. Consider the following example with bytes (0x4, 0x9, 0x3, 0x1, 0x2, 0x7, 0x6, 0x6, 0x5):

1. Add them up and get 0x2B
2. Drop the carry (none in this case) 0x2B.
3. Verify the checksum by adding its 2's complement (0xD5) back to the bytes.  $0xD5 + 0x2B = 0x100$
4. Drop the carry  $0x100 \rightarrow 0x00$ , if it's 0 then the checksum is still good.

Of course, this is a super simple checksum and won't detect errors caused by rearranging the bytes (among other things). Most systems use more sophisticated checksum routines (MD-5 and SHA are two of the more famous ones).

## 3.2 I/O Controller and DMA

In a more modern setting, the hard drives are going to be controlled directly by a controller card, and the CPU (and therefore the OS) will talk to the controller. In this scenario, the OS will request a block from the disk controller and use DMA to transfer the bytes directly into main memory. Therefore the CPU only has to be signalled when the transfer is done or if there's some kind of problem. In this situation, the controller has to deal with all the nitty gritty details. For instance, because the CPU is so much faster than the HD, it's very possible that the OS can request multiple blocks before the controller has serviced the first request. This is actually (potentially) a good thing. Any controller worth its salt will reschedule the requests in order to more efficiently use the hard drive. All the disk scheduling algorithms that used to have to run on the CPU now run on dedicated controller hardware.

## 4 Disk Scheduling

The processor is much faster than the disk, so it's highly likely that there will be multiple outstanding disk requests before the disk is ready to handle them. Because disks have non-uniform access times, re-ordering disk requests can greatly reduce the latency of disk requests. For instance, two requests to the same track are much faster to process in sequence than requests to different tracks. Similarly requests to nearby tracks are faster than requests to distant tracks. A crafty OS will

then try to reduce latency and increase disk throughput by reordering disk requests to best utilize the disk. Following are descriptions of several disk scheduling policies. Performance of these algorithms is usually measured against so-called **random scheduling** where requests are selected randomly from the queue of outstanding requests. To explain these algorithms we're going to use the example of a disk with 200 tracks, and the read/write head starts at track 100. The request queue, in order, contains requests for tracks: 55, 58, 18, 90, 160, 38

## 4.1 FIFO

With a FIFO scheduler, jobs are processed in queue order. If a process is kind and accesses tracks nearby, FIFO may not perform too badly. Even if a process is being nice to the disk scheduler, if there are many running processes, their requests are going to be interleaved, which will mess up the nice process-local ordering.

| Track Number | Tracks Traversed |
|--------------|------------------|
| 100 → 55     | 45               |
| 55 → 58      | 3                |
| 58 → 18      | 40               |
| 18 → 90      | 72               |
| 90 → 160     | 70               |
| 160 → 38     | 122              |
| Average      | 59.67            |

## 4.2 SSTF

SSTF stands for shortest seek time first. This algorithm is based on the observation that seek times are lower for nearby tracks. So SSTF picks the request from the queue closest to the current read/write head location. The only tricky part is if there are two jobs with the same distance (one would be towards the spindle, the other towards the edge). In this case, some kind of tie-breaking needs to be employed to pick one. For instance, you could just use a random number to effectively flip a coin and pick one.

| Track Number | Tracks Traversed |
|--------------|------------------|
| 100 → 90     | 10               |
| 90 → 58      | 32               |
| 58 → 55      | 3                |
| 55 → 38      | 17               |
| 38 → 18      | 20               |
| 18 → 160     | 142              |
| Average      | 37.33            |

As you can see, this is a marked improvement over simple FIFO. However, if a process requests many nearby tracks it can dominate disk activity and greatly increase the latency for other processes

(whose tracks are more distantly located). This condition is known as starvation, because one process is preventing the other processes from accessing the disk (starving them from disk access). This may be optimal, however it is not fair to the other processes. Fairness dictates that the latency be as evenly spread among running processes as possible. So other algorithms have been developed to prevent a processes with high track locality from starving the other other processes.

### 4.3 SCAN/LOOK

Starvation is a bad thing, so OS developers devised a scheduling algorithm based on the elevator algorithm. SCAN services tracks in only one direction (either increasing or decreasing track number). When SCAN reaches the edge of the disk (or track 0), it reverses direction. LOOK is the obvious optimization of having the read/write head reversed when the last track in that direction is serviced.

| Increasing   |                  | Decreasing   |                  |
|--------------|------------------|--------------|------------------|
| Track Number | Tracks Traversed | Track Number | Tracks Traversed |
| 100 → 160    | 60               | 100 → 90     | 10               |
| switch       |                  |              |                  |
| 160 → 90     | 70               | 90 → 58      | 32               |
| 90 → 58      | 32               | 58 → 55      | 3                |
| 58 → 55      | 3                | 55 → 38      | 17               |
| 55 → 38      | 17               | 38 → 18      | 20               |
| switch       |                  |              |                  |
| 30 → 18      | 20               | 18 → 160     | 142              |
| Average      | 35.0             |              | 37.33            |

LOOK behaves almost identically to SSTF, but avoids the starvation problem of SSTF. This is because LOOK is biased against the area recently traversed, and heavily favors tracks clustered at the outermost and innermost edges of the platter. LOOK is also biased towards more recently arriving jobs (on average).

### 4.4 C-LOOK

C-LOOK (circular LOOK) is an effort to remove the bias in LOOK for track clusters at the edges of the platter. C-LOOK basically only scans in one direction. Either you sweep from the inside out, or the outside in. When you reach the end, you just swing the head all the way back to the beginning. This actually takes advantage of the fact that many drives can move the read/write head at high speeds if you're moving across a large number of tracks (e.g. the seek time from the last track to track 0 is smaller than you'd expect and usually considerably less than the time it would take to seek there one track at a time).

| Increasing   |                  | Decreasing   |                  |
|--------------|------------------|--------------|------------------|
| Track Number | Tracks Traversed | Track Number | Tracks Traversed |
| 100 → 160    | 60               | 100 → 90     | 10               |
| 160 → 18     | 142              | 90 → 58      | 32               |
| 18 → 38      | 20               | 58 → 55      | 3                |
| 38 → 55      | 17               | 55 → 38      | 17               |
| 55 → 58      | 3                | 38 → 18      | 20               |
| 58 → 90      | 32               | 18 → 160     | 142              |
| Average      | 45.67            |              | 37.33            |

## 4.5 N-LOOK, F-LOOK

N and F LOOK were designed to offset LOOK's bias towards recent jobs. Both algorithms partition the request queue into smaller subqueues and process the subqueues in order (oldest first). N-LOOK is so-called because the request queue is divided into N subqueues. F-LOOK is a simplification where there are only 2 queues, but they are used in a double-buffered fashion. While F-LOOK is processing one queue, all new requests go into the other one. For this example, we assume that the request queue is split into two, with the oldest one containing the requests for tracks: 55, 58, 18, 90. In this instance, N-LOOK and F-LOOK behave the same. Also notice, that in this configuration, it doesn't matter which direction the head was moving in, all requested tracks are less than 100 so it will only move in the direction of decreasing tracks.

| Track Number     | Tracks Traversed |
|------------------|------------------|
| 100 → 90         | 10               |
| 90 → 58          | 32               |
| 58 → 55          | 3                |
| 55 → 18          | 37               |
| Queue Switch     |                  |
| Direction Switch |                  |
| 18 → 38          | 20               |
| 38 → 160         | 122              |
| Average          | 37.33            |

Even though the average number of tracks traversed is the same as LOOK in the worst case, N and F LOOK are in some sense, more fair than plain old LOOK. The subqueue system caps the maximum latency a process can expect between a request and it being serviced (unlike SSTF that can starve processes for arbitrary lengths of time).