Persistence

> We want to have information on our systems persist, even when the computer gets powered off or crashes.

IO

Systems that don't take input aren't usually very interesting (since they produce the same output always), nor are programs that produce no output (because there'd be no point to running them). This motivates IO.

System Architecture

The CPU is connected to system components via a series of *buses*, with ones (physically) closer to the CPU being faster. Main memory is connected via the *memory bus*, high performance IO devices (like a graphics card) via the *IO bus*, and lower performance peripherals (like disks) via the *peripheral bus* (each getting further away). On modern systems there are quite a few interfaces to the CPU, tailored to the performance requirements of different hardware.

A Canonical Device

Devices have two components: the *interface* they present to other components in the system, and the *internal structure* that enables this abstraction. For example, a device interface could be three registers to view the current device status, initiate commands, and pass data in and out. The flow of using such a device to load in data would involve the OS polling till the device is ready, loading the data, and setting a command register to let the device know.

Using Interrupts

Polling is inefficient, though, so we replace them with interrupts. The OS would instead issue the request, put the calling process to sleep, and the device will raise a *hardware interrupt* when it's done executing, triggering an *interrupt handler* and waking the process.

In general, this approach is quite a bit better because it allows computation and IO to overlap. However, for fast devices where the request would be satisfied by the first time polling, it can actually be slower. To combat this, a *two-phased* approach which first polls and then switches to interrupts can be used. One other performance optimization is *coalescing*, where interrupts aren't immediately

delivered to the CPU in the hopes that more interrupts will come in, allowing multiple to be handled at a time.

Another downside is the possibility of live-lock: we don't want the OS to constantly be stuck serving interrupt handlers rather than actually running user-level processes. We can avoid this by periodically switching between polling and using interrupts as well.

More Efficient Data Movement With DMA

Another source of inefficiency in the canonical model is that the OS spends a lot of time copying data into the memory of the device. A solution is *direct memory access (DMA)*. The CPU just needs to tell the DMA engine what data to copy and it'll do it, raising an interrupt when it's done, thus leaving the CPU free to run something else.

Methods of Device Interaction

There are two primary ways the OS communicates with devices. The first is explicit IO instructions, where the user specifies a register to send or receive data from, and a *port* which names a device. Another approach is to use *memory-mapped IO*, where device registers can be written to as if they are normal memory locations. Neither option is significantly better.

Device Drivers

Another problem the OS could face is that it needs to communicate with many different devices, like different types of disks. Ideally, we want uniform protocol for doing so, so specific code doesn't have to be written for each device. This motivates an abstraction called the *device driver*, which allows the OS to issue uniform read/write calls which are then passed through to devices in a neutral way.

Hard Disk Drives

Hard disks provide a straightforward interface: there are many sectors (512-byte blocks) that we can read from and write to. Writes within a single sector are generally guaranteed to be atomic, and accessing blocks that are near one another in the address space, or blocks in a contiguous chunk is generally much faster than random access.

Components

- Data is stored on a *platter*, and modified magnetically. Platters are double sided, and there can be more than one, connected to one *spindle*.
- Data is encoded in concentric circles of sectors called *tracks*.
- Reading and writing is done by *disk heads*, and there is one of these per surface.
- Each disk head is connected to a *disk arm*.

A Simple Disk Drive

Consider a simple disk drive, with a single 12-sector track that rotates counter-clockwise.

- *Rotation delay* is the time spent waiting for the desired sector to rotate under the disk head. Since the disk rotates in one direction, the worst case requires nearly a full rotation.
- For disks with multiple tracks, *seek time* is the time spent positioning the head on the correct track. It's one of the most costly operations because of the precision required.
- Because outer tracks have more room than inner ones, they usually have more sectors. Tracks with the same number of sectors are grouped into *zones*.
- Disks keep a *cache* to quickly service requests.

IO Time

The time to complete an IO request is the sum of the time spent seeking, rotating, and transferring data. Because time spent seeking and rotating is much larger for random workloads than sequential ones, we can observe 200-300x speed differences based on access patterns.

Disk Scheduling

Because IO is generally costly, disks have their own scheduler to try and complete sets of requests as fast as possible. This allows the OS to issue a series of requests which can then be scheduled using internal disk information that the OS doesn't know. The general principle is *shortest job first*. When developing a policy we have to be careful to avoid *starvation* where we ignore all requests that are far away. This is done by *sweeping* from the outside to the inside of a disk, queueing requests for tracks it's already passed until the next sweep. Other policies consider rotation time as well.

Policy: Shortest Seek Time First (SSTF)

SSTF completes requests from the IO queue that lie on the nearest track. One problem is we don't know the geometry of the disk exactly, but this can be generally approximated by picking jobs with the nearest block addresses. The bigger problem is that it can lead to *starvation*. Given an steady stream of jobs on the same track, we might never server ones that are further away.

Policy: Elevator (SCAN, C-SCAN)

To avoid starvation, we can *sweep* from outer to inner tracks, thus queuing jobs that come in for the current track for our next sweep (rather than serving them immediately).

A downside that persists, however, is that we don't take *rotation* into account. This matters since rotation delays can be about as significant as seek delays, so we can take total positioning time into account (SPTF).

Other questions

There are a series of other potential things that we can consider:

- The OS doesn't have full information, so it makes sense to do at least some scheduling on the disk if possible.
- Multiple IO request can be merged into one, decreasing overhead
- IO requests can be held rather than immediately sent do disk, allowing potentially better requests to come in

RAIDs

RAIDs are *redundant arrays of inexpensive disks*. Depending on how you configure them, using multiple disks can enable improved *performance*, increased *capacity*, and improved *reliability* (by replicating data). These advantages can be enjoyed *transparently* – without having to change anything on the host system.

Interface and Internals

To a file system, RAID is indistinguishable from a normal disk. It looks just like an array of blocks, and when it receives a (logical) IO request it calculates which disks can be used to fulfill it, and issues the physical IO requests. The way these requests are resolved and made depends on the RAID configuration.

Fault Model

To a greater extent than single disks, RAIDs are able to detect and recover from failure. For simplicity, we'll first assume a fail-stop model (a disk is working or it's failed and not running), but failures can be a fair bit messier (for example, data corruption).

Evaluating a RAID

There are three main axes that we can evaluate a RAID along:

- Capacity: how much useful space is available, relative to the total number of blocks?
- Reliability: how many disk faults can be tolerated?
- Performance: how quickly is a single request satisfied? What's the *steady-state throughput* of the RAID (total bandwidth of concurrent requests)

RAID Level 0: Striping

In striping, distributes files across disks in a round-robin fashion, thus maximizing the number of requests that can be made in parallel when serving sequential accesses. Even in random access workloads, striping simply redirects requests to a single disk and thus performs as well as any single disk would.

One small further modification we can make is *chunking*, where the data is striped but in chunk sizes greater than 1. The tradeoff is between request parallelism and positioning time, since the positioning time for a full request is the maximum of the positioning times of the requests. For large chunks this max is over a smaller number of disks.

Capacity is maximal, since every block on every disk is remains addressable. Thus, striping offers strong performance and good capacity, but suffers on the reliability axis: if a single disk dies, data is lost.

RAID Level 1: Mirroring

RAID 1 improves the reliability of RAID 0 by mirroring data across disks, making it possible to tolerate disk failure without losing data. For example, at a mirroring level of 2, every write is made (in parallel) to two distinct disks. Thus, performance is slightly worse, capacity is decreased as a factor of our mirroring level, but we're able to tolerate any disk failing without data loss (and potentially more, if we're not unlucky).

However, the maximum bandwidth in both sequential reads and writes is approximately halved (assuming each piece of data is mirrored once). This is easy

to see, since the upper bound on the number of requests that can be served in parallel is the number of distinct disks that are able to accept IO requests.

RAID Level 4: Parity

We can recover some of the storage space lost by RAID 1 by sacrificing some performance to use *parity* disks. By taking the XOR of data across each of our disks and storing it, in the event of a disk failure we can recover the data. The RAID just maintains the invariant that the number of 1s in any row is even (and so their XOR is 0), and then on disk failure we can recover lost data by taking the XOR of all data and parity bits of surviving disks.

Capacity and reliability are easy to gauge:

- Capacity in this configuration near optimal, especially as the number of disks being used increases, since just one disk is used to track parity.
- Exactly one disk failure can be tolerated before data can't be reconstructed.

Sequential reads can be directed to any disk but the parity disk, giving near-optimal performance. Random reads are also near-optimal, for the same reason. Writes are slightly more complicated by the fact that we have to keep the parity disk up to date. If an entire stripe is being updated at a time, the XOR can simply be calculated and set on the parity disk. Otherwise, there are two methods:

- Read all the other blocks in the stripe and XOR them (*additive parity*)
- New parity bits can be computed as $(C_o \wedge C_n) \wedge P_o$

The first method scales poorly as the number of disks in the system increases, but in some cases is less expensive.

One important downside to this configuration is that *all writes must happen serially*, since they all must update the parity disk. This is particularly costly for random writes, which on average mutate a small amount of data.

RAID Level 5: Rotating Parity

A solution to this last problem is to store parity across disks, rather than on a single one. This removes the bottleneck, primarily improving random write performance by allowing them to be parallelized.

RAID Summary

- Striping is best for pure performance (at the cost of reliability)
- Mirroring is best for random IO performance and reliability (at the cost of capacity)

- Rotating parity is the best for capacity and reliability, at the cost of performance on small writes.

Beyond these things, we could also consider more realistic fault models, more sophisticated recovery techniques, etc.

File System Implementation

The disk is divided up into fixed-size *blocks* (4KB each, for example), and are addressable somewhat like a large array. On disk, we obviously have to store file data, but additionally must keep track of metadata for these files. Lastly, we have to have an on-disk structure (like a bitmap) for tracking what space is free and used. A final block is reserved for the *superblock*, which can track miscellaneous other information.

The inode

Metadata is stored in _inode_s (index node), and can be located on disk using an inode bitmap. An inode tracks information like:

- type: is it a file? directory? ($\neq$ filetype)
- size
- number of blocks
- protection information
- time information

Also, the inode points to the segments on disk where its data lies. Since there's only so much space in the inode, we can't just store a set of direct segment numbers without significantly bounding possible file size.

Multi-level Indexing

To get around this, we can add indirection. For example, we could have the last segment be another table of segments, rather than raw data. This can be extended even further, with double or triple indirection. Each additional layer dramatically increases the upper bound on possible file sizes, at the cost of requiring more indirections to access this data.