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 busy*, high performance IO devices (like a graphics card) via the *IO bus*, and lower performance peripherals (like disks) via the *peripheral bus*. 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*.

In general, this approach is quite a bit better. 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.

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 really 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) from which we can read and write. 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.

Other questions

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

- 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* (since data is replicated). 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. When it receives a request, it calculates which disks can be used to fulfill it, and issues the physical IO requests.