Storage Devices

- Typical Characteristics of Magnetic Disk:
  - 1-5 platters, magnetically coated on each surface.
  - Actuator arm positions heads radially.
  - Roughly 200,000 tracks per radial inch (today's technology)
  - Overall disk package: 1-8 inches in height.
  - Typical capacities today: 100GB-2TB.
  - Typical rotational speeds: 5000-15000RPM. Larger disks tend to be slower.
  - Typical seek time: 2-10ms
  - Average rotational latency (for half a revolution): 4ms (@7500RPM)
  - Transfer: read or write data as it passes under the head. Typical transfer rates: 100-150 MBytes/second
  - Disk API:
    - read(startSector, sectorCount, buffer)
    - write(startSector, sectorCount, buffer)
  - Direct Memory Access (DMA): device can copy data to and from memory, without help from the CPU. This is the common case.
- Flash Memory:
  - Solid state (semiconductor) storage, but non-volatile
  - No moving parts, hence more shock-resistant
  - Faster access than disk
  - 5-10x more expensive than disk
  - Two methods of implementing: NAND and NOR. NAND more popular today.
    - Each device divided into blocks, which are subdivided into pages.
    - Typical block size: 16-256KB
    - Typical page size: 512-4096B. Page is the unit of reading/writing
    - Total capacity: up to 16GB per chip
    - Two significant quirks:
      - Before rewriting a page, its entire block must be *erased*; this is a separate operation. This makes random-access updates expensive.
      - Wear-out: once a block has been erased about 100,000 times, it no longer stores information reliably.
  - Ideal Usage of flash:
    - Write entire blocks (do not update individual pages). Otherwise, writes too slow, wears out blocks too quickly.
    - Wear leveling: manage device so all blocks get erased at roughly the same rate. Avoid hotspots.
- Older method: Magnetic tape
  - 9 tracks acrosss tape (one byte plus parity)
  - 0.5 inch wide by 2400 feet long
  - Typical bandwidth: few Mbytes/sec

Disk Scheduling

- If there are several disk I/Os waiting to be executed, what is the best order in which to execute them? The goal is to minimize seek time. Some options:
  - First come first served (FCFS, aka FIFO): simple, but nothing to optimize seeks.
  - Shortest seek time first (SSTF):
    - Choose next request that is as close as possible to the previous one.
    - Good for minimizing seeks, but can result in starvation of some requests.
  - Scan or elevator algorithm:
    - Choose a direction (inwards or outwards) based on FIFO order
    - Keep moving in the chosen direction, reading the blocks on the way, till you reach the last block. In other words, use SSTF for all blocks that are accessible in the chosen direction.
    - If no more blocks in the current chosen direction, change direction.
    - Similar to how an elevator works typically (replace inwards/outwards with up/down).

UNIX Filesystem Interface

We will focus on magnetic disks, given their popularity. The interface design and implementation heavily depends on the characteristics of the underlying storage device. Most OSes today, have been optimized for magnetic disks.

The API for a minimal file system consists of: open, read, write, seek, close, and stat. Dup duplicates a file descriptor. For example:

```
fd = open("x/y", O_RDWR);
read (fd, buf, 100);
write (fd, buf, 512);
close (fd)
```

Notice that this interface assumes that the file is a stream of bytes. Alternatively, the data in a file could have been organized in a structured way.

The structured variant is often called a database. Any particular structure is likely to be useful to only a small class of applications, and other applications will have to work hard to fit their data into one of the pre-defined structures. Besides, if you want structure, you can easily write a user-mode library program that imposes that format on any file (with performance penalties however because of layering of different abstractions one above another). (Databases have special requirements and support an important class of applications, and thus have a specialized plan.)

To allow users to remember where they stored a file, they can assign a symbolic name to a file, which appears in a directory. There's a couple of different things going on here.

- Names: how do we get from the name "x/y" to the file we're actually talking about.
- Directories: provide a way of assigning user-meaningful names to files. This is a part of the kernel name interface.
- In Unix (and almost all other filesystems) there's some notion of an on-disk file that's independent of the file's name, called an "inode". Seen by the user (e.g. stat) but cannot access a given inode.
- Disk blocks: the inode translates offsets in our conceptual file into concrete locations on disk. Not seen by the user.
- File descriptors: in Unix, the FD binds to the inode, rather than the pathname, so the pathname lookup is done only when the file is first opened. Application handle for an inode in the Unix API.

Maintaining the file offset behind the read/write interface is an interesting design decision. The alternative is that the state of a read operation should be maintained by the process doing the reading (i.e., that the pointer should be passed as an argument to read). This argument is compelling in view of the UNIX fork() semantics, which clones a process which shares the file descriptors of its parent.

With offsets in the file descriptor, a read by the parent of a shared file descriptor (e.g., stdin) changes the read pointer seen by the child. This isn't always desirable: for example, consider a data file, in which the program seeks around and reads various records. If we fork(), the child and parent might interfere. On the other hand, the alternative (no offset in FD) would make it difficult to get "(echo one; echo two) > x" right. Easy to implement separate-offsets if kernel provides shared-offsets (re-open file, mostly), but not the other way around.

The file API turned out to be quite a useful abstraction. Unix uses it for many things that aren't just files on local disk, e.g. pipes, devices in /dev, network storage, etc. Plan9 took this further, and a few of those ideas came back to Linux, like the /proc filesystem.

Unix API doesn't specify that the effects of write are immediately on the disk before a write returns. It is up to the implementation of the file system within certain bounds. Choices include (that aren't non-exclusive):

- Before the write returns [most conservative, but bad for performance];
- Before close returns [AFS];
- At some point in the future, if the system stays up (e.g., after 30 seconds) [highest performance];
- Application specified (e.g., before fsync returns) [requires careful application coding];
- Before external things (screen, network) indicate the write returned [will read paper about this later].

Some examples of file systems:

- Contiguous Allocation: `file a = (base=10, len=12)`
  - Pro: Fast sequential access, easy random access
  - Con: External Fragmentation/hard to grow files
- Linked-list Allocation: `file = linked list of blocks`
  - Pro: Easy to grow file. Free list can be implemented as another file.
  - Con: Bad sequential access performance! Unreliable: Loose block, loose rest of file
  - Serious Con: Bad random access