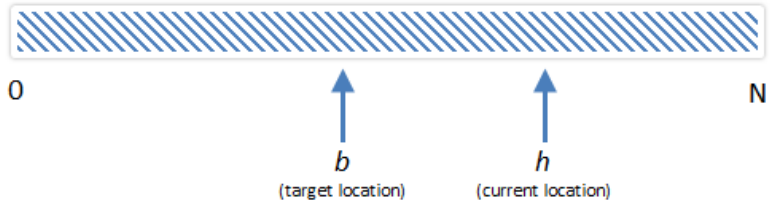
[CS 111] Spring '12 Scribe Notes - File Systems

By: Michael Gendler, Ben Phung, and Allen Wu (from Lecture #12, 5/10/12)

1. Simple Hard Disk Model
2. Disk Scheduling Algorithms
   a. First Come, First Serve (FCFS)
   b. Shortest Seek Time First (SSTF)
   c. Elevator
   d. Circular Elevator
   e. Anticipatory (Speculative)
3. File System Design
   a. Prof. Eggert's file system (~1974)
   b. File Allocation Table (FAT) file system (~1980)
   c. UNIX file system (~1975)
   d. Berkeley Software Distribution (~1977)

Simple Hard Disk Model



As a model, we can map the multi-layered, two-dimensional platters that make up the hard-drive to a linear, one-dimensional array. We can imagine that both the disk head and the data of interest are tracked by two respective pointers in this one-dimensional array. "b" is the data location while "h" is our current location on disk.

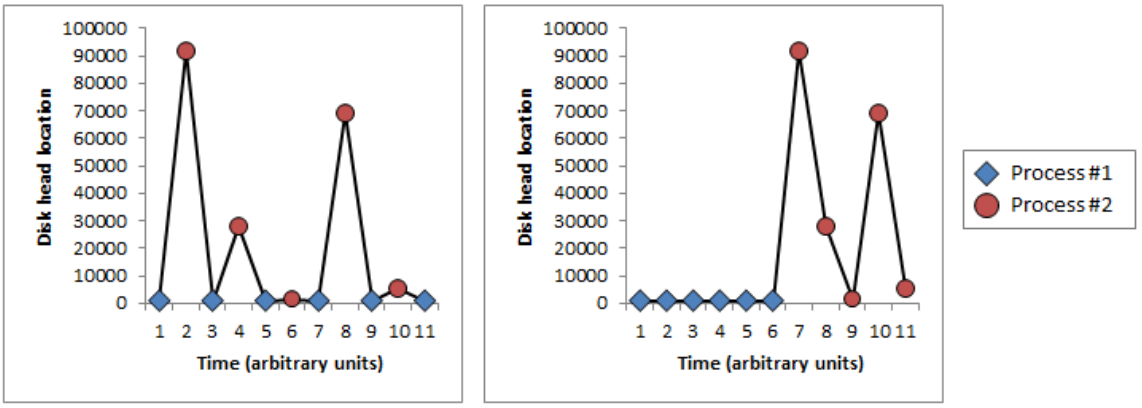
By this logic, the cost of a seek is $|b\text{-}h|$. This approximation tends to overestimate long seeks and underestimate short seeks due to the nonlinear nature of the disk head movements; in particular, acceleration time becomes more negligible with larger distances, but maintains a more prominent role in shorter distances.

Disk Scheduling Algorithms

First Come, First Serve (FCFS)

First come, first serve means we queue and service requests in order. Assuming that we are undergoing random access, and that both b and h are random, what's the average cost of an access?

Let's take a look at an example. Assume process #1 and process #2 wish to concurrently read data from the disk. Process #1 wants to read blocks 1000, 1001, 1002, 1003... (sequentially located blocks). Process #2 wants to read blocks 91631, 28193, 1572, 69317 (random, non-contiguous blocks).



The graph on the left shows process #1 and process #2's interleaved seeks using FCFS. Notice the large amount of disk head movement during every time interval. The graph on the right shows a different approach. All of process #1's

requests are completed before starting process #2's. The result is that during process #1's execution, the disk can utilize sequential reading, which is much faster than the random seeking that process #2 requests.

Shortest Seek Time First (SSTF)

SSTF will always schedule the request that is closest to the disk head.

- ✓ Allows better throughput because of minimal seeks
- ✗ Causes starvation. Consider the following analogy: an elevator continually moves to service its closest request. If the Boelter Hall elevator worked like this, it would be continually tending to 4th and 5th floor requests, leaving 9th floor requests to starve.

Elevator

The elevator algorithm is the same as SSTF, except that the head will service the closest request that can be reached when moving in a particular *direction*. In other words, the Boelter elevator that has just moved up from the 4th to the 5th floor will continue servicing requests all the way up to the 9th floor, while ignoring new requests from the bottom 4 floors. Once it reaches the highest requested floor, the elevator will make its descent, servicing floor requests in descending (SSTF) order.

A drawback of the elevator algorithm is that the edges (floors 1 and 9) have an average wait time of 1 cycle, while the middle floors have an average wait time of 0.5 cycles. (The middle floors get serviced twice: once on the descent, and once on the ascent).

Circular Elevator

The circular elevator algorithm connects both edges together as if the head is always moving in a particular direction. In this case, the Boelter elevator will reach the 9th floor, immediately drop to the 1st floor, and then service floor requests in ascending order. This scheduling scheme is fair to all processes but comes at a performance cost - resources are being idly wasted during the "drop" portion of the algorithm.

Anticipatory (Speculative)

In the anticipatory algorithm, we want to design the file system such that it will speculate on what a process will do and delay (DALLY).

From the example, envision a scenario where we just read the first thousand blocks in a process. After finishing the read, there is only 1 (far) read request that is pending in the system. Instead of servicing that request, the algorithm would delay for a particular unit of time (0.1 milliseconds) in hopes of a request coming in that would be cheaper to service.

However, starvation is still an issue. If, for example, process 1 had a very long queue of sequential blocks, then process 2 would succumb to starvation. To reconcile this issue, the schedule typically keeps track of long-waiting requests with a timer to ensure that they too get serviced.

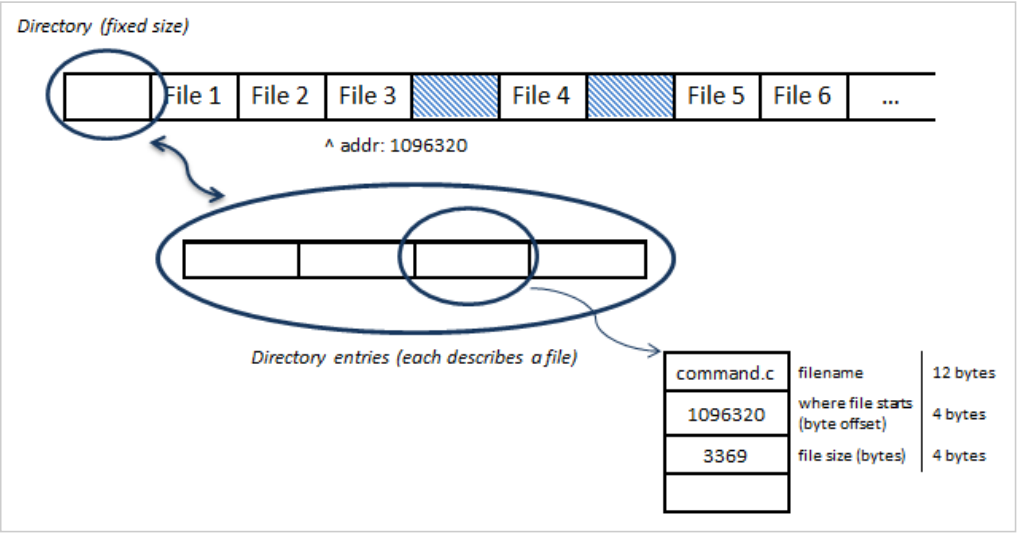
File System Design

File systems require a data structure that lives on disk and provides an abstraction (illusion) of a set of organized files. Modern file systems span many different physical devices, including hard disks, flash-based storage, and network components (local or cloud servers). Each of these components has a performance limiting factor.

Controlling factors of component performance
- Disk: seek time
- Flash-based storage: I/O per second
- Network: network latency/throughput

Prof. Eggert's file system (~1974)

> "...before he knew any better!"



This file system is similar to an array where each file starts on a sector boundary. It turns out that this file system was similar to RT-11, an existing file system.

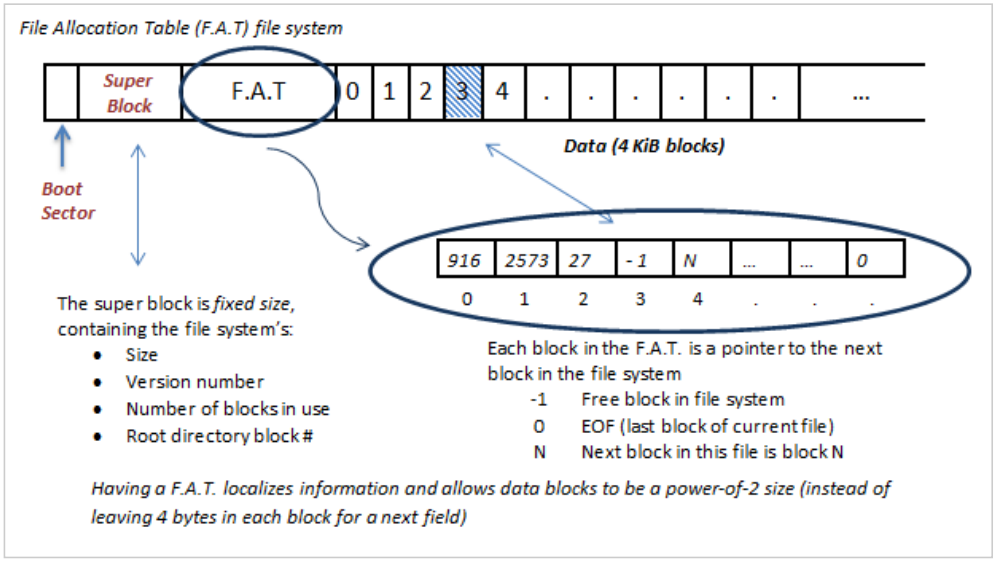
Advantages:

- ✓ simple
- ✓ sequential accesses are fast because all files are in order (disk arm doesn't have to move)

Disadvantages:

- ✗ have to specify its size when you create a file (worst limitation)
- ✗ external fragmentation (*total* free space > *requested* free space, but *contiguous* free space < *requested* free space)
- ✗ internal fragmentation (`size%512` is wasted in each file, assuming 512 byte file size)
- ✗ must know the maximum number in each file when creating the file system

File Allocation Table (FAT) file system



Boot Sector

The first sector is at index 0 and includes the BIOS Parameter block and boot loader.

Superblock

The superblock has a small, fixed size and contains the file system's version number and size, the number of blocks in use, and the root directory's block number.

File Allocation Table (FAT)

An array of block numbers. The FAT holds integers where each one is a pointer to a block number in the data area. It differs from the previous file system as the files are no longer contiguous.

The integer can be:

`-1`: indicates a free block

`0`: indicates end of file

`N`: indicates the number of the next block in the file

Directory

Name (11 bytes)	Size in bytes	Block # of first block

8 bytes for name
3 bytes for extension

The directory is a file that contains directory entries. Each file will contain:

- a name with 11 bytes for the file name (8 for the name, 3 for the extension)
- size in bytes
- the block number of the first block

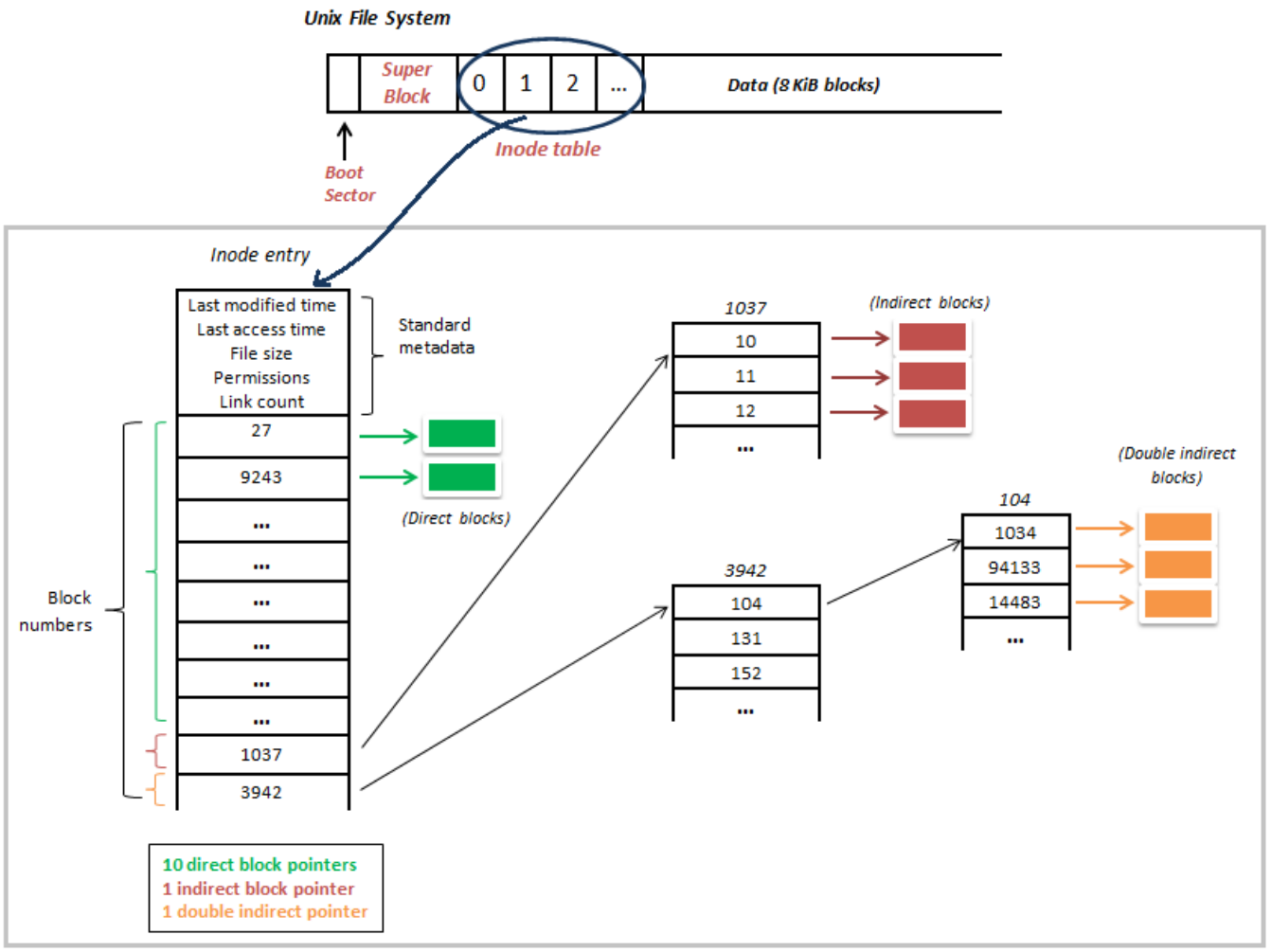
Advantages of FAT:

✓ easy to grow file
✓ no external fragmentation
✓ no artificial limit to # of files

Disadvantages of FAT:

✗ not robust if disk is flaky (i.e. if the directory messes up, we are toast)
✗ sequential reads can require lots of seeking (but the defragmenter can fix this)
✗ `lseek` requires $O(N)$ of time

UNIX file system (~1975)



Boot Sector

The boot sector contains information to be loaded into RAM in order to boot up the OS.

Superblock

This holds the file system's metadata information such as file system type, size, status, and other information about other metadata structures. Problems with the superblock can cause a lot of trouble, which is why OS's like Linux hold multiple redundant copies of this.

Inodes

This system is based on inodes (internal nodes). Each inode is a file descriptor containing metadata which holds the file's size, permissions, owner, links, and timestamps. One can use `stat()` to find out more on this. In addition, inodes do not contain the file name nor the directory containing it.

Each inode may contain 10 direct block pointers, 1 indirect block pointer, and 1 doubly indirect block pointer.

Direct block pointer

points directly to data blocks

Indirect block pointer
: points to a block of direct block pointers (2000 block pointers in a file system with 8 KB blocks and 4 byte block pointers => 16 MiB)

Doubly indirect block pointer
: points to a block of indirect block pointers (2000 indirect block pointers which means 32 GiB (2000x2000x8 KB) of data!)

Using indirect and doubly indirect block pointers, the inode can link to a file's data whose size is a lot larger than the amount it would be able to hold with just direct block pointers.

Each file is linked to an inode which also keeps count of the number links to the file. This allows for hardlinks using the command:

```
$ ln a b
```

And even if we do:

```
$ rm a
```

file b will still be accessible because the link count will still be greater than 0.

Holes
: The major issue with this file system design is the problem of holes. Holes are basically deallocated space and are read as 0.

  To create holes one can use

  ```
  $ dd if=/etc/passwrd of=big seek=10T
  ```

  To see the block size and number of block used, one can then use

  ```
  $ ls -l big
  ```

  The problem with holes will be when the file is being copied such as

  ```
  $ cat big
  ```

  This will result in a extremely large amount of data blocks full of holes being `cat`'ed.

Berkeley Software Distribution (BSD)

The Berkeley file system is extremely similar to the UNIX file system, but it also adds a block bit map. This allows the file system to know whether a data block is free.

BSD still succumbs to external (similar to FAT) and internal fragmentation.