CS162 Discussion #09: Disks and File Systems

Announcements

- Midterm regrade requests due TODAY
- If your project 1 doesn't work…come to office hours this week! It needs to work for project 2.
- Project 2 code due next Thursday 11/03 @ 11:59pm (`submit proj2-code`)
    - Autograder will be up with `submit proj2-test`
    - Final design docs & group evals due 11/04 @ 11:59pm

If you'll be my disk, I'll be your diskette

Last week with demand paging, we talked about "writing pages to disk" and "reading pages out of disk", but we haven't actually talked about what a disk is. A hard disk is a non-volatile storage device: it can retain data without power. A disk has the following components:

- *Platters*. A number of rotating circular platters, positioned in a column.
- *Read-Write Heads*. Heads are located above/between the platters, and read or write information to them using magnetization.

Here are some different ways we can look at the regions of the disk:

- *Track*. A fixed-radius circular band on a platter.
- *Cylinder*. The group of tracks over all platters corresponding to some fixed radius.
- *Sector*. A region of a track. This is the smallest independently addressable unit on the disk – each sector on the disk contains the same number of bits.
- *Block*. A group of sectors to be transferred together.

The rotation of the platters allows the head to read data off of the disk as the data passes by, by sensing the magnetization on the platter below the head. Therefore, without moving the head, we have access to an entire cylinder of information. We can change the angle of the arm on which the heads are located to change between tracks; this is called seeking. This combination of seek and rotation allows us to cover any part of the disk we need to.

Note that if we take sectors in the strictly mathematical sense (i.e., each sector is a region between two fixed angles from the center of the platter), we get sectors like pie slices, where the sectors on the outer tracks are wider than the ones on the inner tracks. However, since all sectors have to contain the same number of bits, the bits on the inside get squished and have little extra room, while the bits on the outside have a lot of extra room.

We can improve our usage of the disk by using a technique called zoned bit recording, which makes all sectors on the disk the same size so that the bit density stays constant. This means outer tracks will have more sectors than inner tracks. This brings us to another interesting point – since the outer tracks have more sectors using zoned bit recording, the data transfer rate there is higher. No matter how fast the platter is spinning, a greater distance will be traversed by the head on the outer tracks in comparison with the inner tracks within the same amount of time. Therefore, when reading from the outer tracks, we read a larger amount of data in the same amount of time.

Does this make me look FAT?

We just saw that disks contain blocks of data. We can group blocks of data together to make files and directories – this is managed by the file system. The file system is in charge of grouping blocks into files, finding files by user-friendly names, and protecting the data. Each file has a file header, also called an *inode*, that keeps track of overall information about the file; the information stored here depends on how we organize our files. We have a lot of options when it comes to organizing files on the disk, but as always, we want to organize them in the most effective way. When looking for an organizational structure, here are the goals we would like to achieve:

- *Maximize sequential performance.* Since files are read and written sequentially most of the time, it makes sense to make this type of access as easy as possible.
- *Easy random access.* While it happens a little less often than sequential access, it's also important to be able to jump to some desired location in the file without having to go through a lot of bytes we didn't have to read.
- *Easy file management.* We want to be able to grow and truncate our files as necessary without too much trouble.

With these goals in mind, we can consider the following techniques for organization:

- *Continuous Allocation*. Use a continuous set of blocks on disk to store the file. This is good for sequential and random access, but it's hard to change a file's size because the blocks have to be continuous.
- *Linked List*. Each block has a pointer to the next block on disk the rest of the file is located at. Here it's easy to change file size because we can allocate blocks anywhere and just add a pointer to the previous block…but sequential access is now bad because we have to seek between reading each block.
    - *File Allocation Table (FAT)*. A table containing an entry for each disk block, with entries corresponding to the same file being linked together.
- *Indexed Allocation*. The file header points to a block that holds pointers to all the blocks used by the file. Random access here is fast because we know where all the blocks are right away, but it's hard to grow the file past the size that fits in the indexing block. We also still spend a lot of time seeking if we read sequentially, because the blocks can be anywhere.
- *Multilevel Indexed File*. The inode contains a portion of direct pointers to data blocks, a pointer to an indirect block which in turn contains pointers to data blocks, a pointer to a doubly indirect blocks which contains pointers to indirect blocks, etc. This makes small files more efficient and it's relatively easy to grow the file larger up to a certain point, but it gets less and less efficient the larger your file becomes and sequential access still requires a lot of seeking.

In addition to the fact that we have to find a way to organize files on disk, we also have to have a way to allow users to find the files easily – users don't want to have to remember *inumbers* (unique indices associated with inodes), so we have to support naming. In this case, some sort of translation from a name string to an inode is required. This is where directories come in. A directory is basically the same thing as a file, except it contains a mapping from file names to inumbers. This allows us to reuse whatever system we were using to store files. Any named path requires a disk access for each name in the path, each one accessing a directory file until the name is resolved.