CS162 Discussion #11: File Systems

Announcements

- Phase 3 code due Tuesday, 11/16 @ 11:59pm (`submit proj3-code`)
  - Nachos Night will be Monday, 11/15 @ 7:00pm in 711 Soda
- Phase 4 starts Wednesday, 11/17 (Last project! Yay!)

Does this make me look FAT?

Last week we 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 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.

Let's go on a RAID!

One of the guarantees users want from a system is to know that their data is actually safe. Users don't want to think they've successfully made changes to a file and find out later that the changes were never saved, or that changes were only half saved. A fault in the middle of making changes to the file system could leave the system in some intermediate state where some operations have only partially happened.

We can avoid this situation by using a log to record changes to files, as transactions. These changes can then be read from the log to perform the actual file system updates at a later time. Any change that has been written to the log counts as being committed, whether or not the file system as actually been updated yet; if there is a problem halfway through writing a log entry, we ignore the entire entry. This makes changes atomic because based on the log, we can determine whether or not to actually perform the operation. It also increases reliability because writes to the log are relatively quick when compared with the time it takes to traverse the file system structures for every update. Log entries don't get deleted until the update has been completed, so they can be used for recovery if necessary. A system like this is called a *journaling file system*.

Another way we can make our system more reliable is by using *Redundant Arrays of Inexpensive Disks (RAID)*. The basic idea behind any type of RAID is that if we have multiple disks, we can use some of the disks to write redundant data to so that it can be used for recovery in case something goes wrong with the original data. In RAID 1, this consists of mirroring data by writing any data written to one disk also to a second disk. (We can also mirror across more than 2 disks.) Other types of RAID make use of parity blocks which are computed using a combination of the blocks stored on other disks; failure on one disk (or for RAID 6, two disks) can be tolerated because the surviving blocks on the other disks can be combined with parity blocks to recover the lost data.