Review from last time---this was all "last five minutes" material

part 2: representing the filesystem on-disk

what do we need to store? (can be interactive):

- user data (file / directory contents) <--- note, this is where the directory structure goes, it is *not* independently stored anywhere else.
- inode data (metadata / stuff returned by stat)
  - important thing: this is a table of inode numbers, and the *most* essential thing they need to store is the block-address of the data in the real filesystem.
  - *this* is the point where we get to *ask which* inode metadata we should really be storing. We can take a look at the output of **stat** again and come up with thoughts about what the "obvious" stuff to put on disk is.
  - anything else?

At this point, folks will probably *not* have an idea of what else to store. We can segue quickly into how to lay this stuff out on disk

- Open by explaining how to access the disk. Talk about blocks in the filesystem: note that they can be composed of **multiple sectors**. The word "block" when talking about a raw disk just doesn't mean the same thing as "block" when talking about a correctly-formatted filesystem *residing* on this disk. (It can, though, if you want it to!). Sorry y'all.
- Talk about reserving an early portion of the disk for the inode-table, and everything else in the disk for the data.
- Ok, challenge: a file's refcount is now zero; we would like to remove it from the disk. How should we do this?
  - remove both the data *and the inode contents*, recall.
  - **we need some mechanism to now note that this inode / data region is free**
- from here, we can segue into the bitmaps for the inodes and the data region
  - remember slab allocators?? Here's the biggest, most-famous use of them: inode arrays and data arrays on disk are **often** (but not always) slab-allocated.
  - So, we have already found our big region of fixed-size chunks; now we just need the bitmap to tell us what is free in that region!
  - (review the slab allocator semantics again, especially how using bitmaps works)
- ok, are we done? *not quite*. We need one last item in the filesystem: a small amount of metadata *at a known location* that tells the OS where to find all of these regions and bitmaps! This is called the **superblock**. It usually also contains a hash of key information, and an FS identifier telling the OS what FS this is.
  - If you've ever tried to mount a filesystem on linux and gotten an error back of "bad superblock," then that means some of this metadata was corrupted. Note that it does *not* mean that the data in the filesystem itself was harmed! Just that the OS has no idea where to look to find it, or what format it should be in when it does find it.
- another note: we can think about atomicity when we are building and using this. remember: **sector writes are atomic**. So structures where a torn write is very bad ought to stay smaller than a single disk sector. Makes filesystem recovery plausible.

ok, let's try to use this thing

- return to the example directory structure, and let's walk through a (hypothetical) set of disk accesses, using the on-disk and in-kernel structures we've already developed
  - recall: in-kernel inode table, on-disk inode table, superblock, data region.
  - show how we know how to index into... anything.
- if you have time, here are some other things that we should go over in class
  - how do you know how *big* a file is? We talked about an inode containing a pointer to the first block in a file, but what about the last block? There are options here; the three big ones are "just a linked list," "array of block pointers (with indirection)", "it's just segments all over again." Note: we have all the freedom here that we *wished for* in the VM world! There's no hardware to play-nice with, and we have SO MUCH LATENCY to hide things in. It's great!
  - extended permissions: ACLs
  - fun tricks with UNIX permissions (e.g. groups can r/w but users can't). SetUID and setGID binaries. The idea of the "root" user.
  - the rest of the data in the in-memory file table---e.g. current read position and seek/lseek
  - **unified page cache (see the textbook for details about this)**
  - buffering writes (and the associated need to **fsync** after)
  - performance considerations / memory usage trade-offs

Part 3: realistic issues in filesystems

- contiguous access (remember: flash "flashes" big chunks at a time (about 256KB); it's "free" to access neighbors)
- fragmentation
- LVM -- logical volume management. AKA, "why do inodes need to be physical addresses anyway?" lets us solve fragmentation "transparently"
- special large files treatment?
- power loss, or how to keep things sane on the disk in an emergency (recall, sectors are atomic)

Journaling and filesystem recovery

So far, a *lot* of the problems we've seen with disks mirror those that we've seen with memory---virtualized mappings, fragmentation, caching policies, allocation strategies and the like are all basically the same on-disk as they are in-memory. In fact, they're arguably *better* on the disk; because the latency of transfer is our bottleneck, we often* don't need to rely on restricted hardware primitives to implement our filesystem abstractions. So we really *can* just design whatever datastructures we want!

But it also comes with a *major* drawback: the disk is persistent. It will continue to exist after power failures, and we will need to be able to read from it even in that situation.

Let's walk through an example of what happens when we write to the disk under our basic filesystem, and then see what's going on after a power failure (note: we should already have walked through what reading and writing from the on-disk filesystem looks like by now. If we haven't, stop and do that).

So, looking at this, we have a *lot* of metadata. We also have a lot of distinct positions in the disk where we need to update to perform a file allocation. Let's look at what happens if we lose power in the middle of doing one of those. (we do this on the board)

How do we fix this?

Point 1: remember sector atomicity! If we can fit *lots* of stuff in a sector, then the sector *can't* show us a "skewed write". And sectors can be reasonably big---at least 512 bytes (and the atomic region is a lot bigger in flash---many kilobytes). If we make sure that our inodes are sector-aligned (e.g. no inode splits across sectors), then it should *never* be the case that our inodes reflect a split write. Our metadata is safe!

Point 2: Sometimes, some operations can be carefully handled *in a particular order* to effectively guarantee sanity. For example, if we're looking to create a file or change its size, we can do this in the in-RAM inode first, then in the data portion of the disk, and finally in the on-disk inode. Similarly, when creating a file, we can always reserve a slot in the *in-memory* (ram) inode table first, and then make sure the inode is well-formatted on disk, before writing to the on-disk inode bitmap. It's entirely possible to create always-sane filesystems using some combination of these two approaches: ensuring atomic writes, and carefully selecting the order of operations.

In fact, this is a special case of a more general pattern of *lock-free datastructures*. Check it out.

But it also seriously constrains the design space of our filesystems! There's a pair of much simpler solution that's used in practice:

Let it burn.

Just write a tool (usually a version of fsck) that goes through the filesystem and tries to make the internal structures sane, by restoring their data invariants. By combining an fsck-recovery with the idea of being careful about write-order, you can represent a much bigger class of filesystem designs while maintaining crash consistency!

On our simple filesystem, here's how fsck would work (draw this):

- superblock: figure out what we're looking at, and where the inode, bitmap, data tables are. Do some basic sanity checks; the filesystem is of a known type, the overall filesystem size in the superblock is within the bounds of the current disk partition, etc.
  - Note: filesystems often have many *copies* of the superblock, so that corruption of one superblock copy doesn't render the filesystem unusable.
- inodes: rather than relying on the bitmap of free/used, we scan the entire inode table and recreate the on-disk structures *as the inodes see them*. Note that this requires that we have some metadata in the inode that helps us understand if we're looking at garbage data; but even here, sanity-checking helps.
- Data: we look through the filesystem tree on the disk, to see how many links each inode *actually has* in the directory structure. Inodes that don't seem to be linked in anywhere refer to "lost" files, and we can link these in to a special "lost+found" directory at the root of our filesystem (or discard them, if you want).

There are a few common error cases you'll hit with this:

- bad block: an inode pointer is going somewhere obviously impossible
- duplicates: two inodes own overlapping disk regions. We usually deduplicate through copying.
- general corruption: lots of fields can have impossible values. Usually just zero them out.

This kiinda works, but it is **very very** slow.

The filesystem journal.

This is stolen from the databases community, where journaling has been long-studied. The short version: you want to make filesystem operations "transactional," and guarantee atomicity across multiple disk accesses. To do this, we reserve a space on the disk to serve as a *ring buffer*, and always write an "intended action" to that buffer *before* we begin executing it. In addition, we do not start executing the first intended action in that buffer until *all intended actions in a single transaction* have been written to the buffer. *Only then* can we execute the first action. Finally, we need to make sure these transactions are idempotent; doing a logged transaction twice in a row (even if the transaction was only logged once) should always be fine (as long as there were no intervening operations). Ok, now really finally: write a quick "done" message to the log after the main-disk operations have all been flushed to disk.

You need to draw this on the board. You should also just put the journal somewhere on the disk, usually in the front.

The correctness idea is simple; in normal operation, this log just gets continuously appended to and never observed. But during a failure recovery, we can look at the log to see if there's any transactions which have been fully written to the log, but *don't* have the "done" marker after them.

Interactive: given the simple protocol we've described, is there a bound on how many not-quite-done transactions will be in the log?

For the simplest logs (like the one we've described), there will be at most *one* of these transactions present. For more-complex versions, there can be several (e.g., we delay access to the main filesystem for performance reasons so that we can do *write coalescing* pretty often).

Interactive: so, what do you suppose we should do now?

Well, just redo the transaction!

Interactive: what do you think we should do about transactions that are only *partially present* in the log?

Well, this depends on the user-facing semantics you want. The usual answer is that you should just discard them; more advanced answers include that you should run a "recovery tool" which can deduce missing actions based on the parts of the log that are present. Or even that you should create a special "lost+found" directory on the disk, where you can link in files (or file fragments) whose contents appear in the log but whose location in the filesystem is forever lost. Lots of options here!!

What about performance?

Interactive! Do we care? Do we have ideas? Talk to your neighbor.

if we have time, talk about revoke records.

If we have time, talk about the log-structured filesystem (append-only frw!)

- inodes are scattered; *inode map* is written periodically to the end of the log, which points to all of them.
- inode map is still scattered; a single *checkpoint region* points to the latest inode map
- note: both of these things are really only needed for recovery. An in-memory inode map (and inode table) works just fine for regular access. And the log is built-in, so it's really just "replay from the last point the checkpoint-region points to"
- final note: the inode-map is a necessary indirection; inodes can move, so their exact location *cannot* be stored in directories! See also lvm (logical volume management)
- garbage collection btw, it's the usual answer.