Basic filesystems

part 1: the FS interface to userspace

- review again how the filesystem works; that the filesystem is, from the OS perspective, a giant array of **inodes**. The "directory structure" is imposed on it; it's just the contents of **directories**, a specific kind of file.
    - Write out what this looks like for simplified directory structures a hypothetical directory tree. Point out that the whole "files are in one location" is a convention, not a rule. Use the illustrated example to suggest that it's easy to put a file in more than one location
- log in to the cluster, **go to /tmp** (to avoid nfs shenanigans), and build (the equivalent of) that filesystem in some subdirectory of **/tmp**. Use this filesystem to show the idea of files being linked in at different points
- show the **stat** command, and highlight the link count
- review the idea of the "file descriptor" in an open process---remind the class that the directory tree is *one* overlayed structure on the filesystem, but in fact you can have *many* if you would like. An open file descriptor, on UNIX, is just another mapping to an inode.
    - "So you can have an inode linked in at one or several points in the directory tree, *and* you can have an inode associated with a file descriptor in one or several running processes. These are independent. You can close every open file without changing what the filesystem sees; you can also remove (unlink) every directory entry for a particular file *without implicitly closing it*"
- show the class the **stat** command again, so that you can see how many links a file has in the filesystem *after it has been removed* from the directories entirely.
- direct talk notes
    - This is cool, but it begs the question---how do we *ever* actually remove files? We can't rely on its presence in the directory structure to do this, so we'll need some sort of *garbage collection* routine within the filesystem implementation to handle this. In many simple filesystems and operating systems, this takes the form of a **reference count** for the file, counting both the number of links the file has in the directory structure *and* the number of processes that has it open.
- point out that in order to track this, our in-kernel **proc** datastructure will need a table of open files, and we will *also* need an in-kernel table of accessed **inodes** where we can put this reference count. We also put some inode metadata into this datastructure (much of which is revealed by **stat**)
    - note---you'll come back to this, do not do it live---we're going to want to ask ourselves what of this metadata should be reflected on-disk. This can be interactive *when the time comes*
- Ok, one last note: permissions. UNIX files have permissions; where should those live? **This one can be interactive**. Ask the class to propose locations for these permissions. Don't give the right answer, *come up with experiments* to determine whether they are right (e.g. by leveraging hardlinks or permissions in intermediate directories).

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.
    - "Ok, now how do we find this stuff? User files have a bunch of diverse sizes, so we can't just do a "giant array of data" approach here."
    - (Show this issue; take your hypothetical directory structure and just lay it out, and you *immediately* get stuck when you hit an inode that doesn't just-so-happen to be the one immediately after the inode for '/')
    - interactive: how do we index this stuff? Is there another datastructure we could store, or rely on, to find it?
- 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. Heck, we never really talked about what UNIX permissions really look like!
    - the rest of the data in the in-memory file table---e.g. current read position and seek/lseek
    - **unified page cache**
    - buffering writes (and the associated need to **fsync** after)
    - performance considerations / memory usage trade-offs
-