Assignment 4 Section Notes #2

CS161 Course Staff

April 21, 2015

1 Administrivia

- There's a patch to truncate; it's been pushed to the handout repo. See Piazza.
- While there's no class 4/30, we're trying to make sure we have the room reserved so everyone can come and hack in 301 during class time.
- The assignment is due May 1.

2 Unlinked Files

As (hopefully) discussed in designs, you need some kind on-disk morgue / graveyard / orphanage / homeless shelter for holding files that have been unlinked but not yet reclaimed.

There are various possible ways to implement this. The "right" way is to use an extra inode, and either reserve its inode number or store the number in the superblock. If you use an inode, you can use `sfs_metaio` and other already-existing functions to manipulate it. (You can even make it a directory and use the directory functions to manipulate it.)

However, there are other cheesier things you can get away with here that you probably wouldn't want to do if you were writing a file system for production use, like reserve a special block or even use space in the superblock.

In any of these cases, you want to make it a first-class citizen of the file system, meaning

- have mksfs initialize it;
- teach dumpsfs to dump it for debugging;
- teach sfsck to know it's there so it doesn't think the blocks involved are leaked;
- journal all updates to it that occur at run time.

Ideally you also want sfsck to check that it's empty (exercise: why should sfsck always see it empty?) but this is less critical.

To use your morgue / graveyard / orphanage / homeless shelter, you'll want to move files and directories to it when they're unlinked, which is in `sfs_rmdir` and `sfs_remove`. Then when they're reclaimed, in `sfs_reclaim`, take them out again. (Both regular files and directories should be handled this way as both can be held open after being unlinked.)

Recovering with the journal will give you a file system where any leftover unreclaimed files are sitting in the graveyard. Therefore, after this is done, but before mount finishes, you can go through the graveyard and finish reclaiming the files there. This involves both truncating and freeing the inode - note that you don't need to write special code to do this, but can instead just load a vnode and drop the reference that gives you; that will go through `VOP_RECLAIM` in the normal way. Clearing the graveyard will create journal records, because it updates the disk, so after you're done with this you may want to issue another checkpoint before going fully live.

Note that directories are truncated at `rmdir` time, because the semantics of directories require that you can't use them once they're unlinked; but regular files aren't. Consequently some of the objects in the graveyard will need truncation, and some won't; this doesn't really matter though.

3 Implementing Journal Records

Some tips for implementing journal records without going crazy:

- Define a struct for each record type. If you have common pieces, use more structs for those.
- Give each record type a type code. Create #defines for these, or an enum.
- Put all this material in `kern/sfs.h` so it's visible in userland code.
- When reading records out of the journal, switch on the type code first.
- Do something reasonable (panic, fail) if you get an unknown type code. This will likely happen to you at least once due to bugs. Don't ignore unknown records.
- Assert that the size of the record is what it's supposed to be, or within the bounds of what it's allowed to be.
- Only then touch the blob pointer.
- Unless you're *sure* your records are always fully word-aligned, memcpy out of the journal into an instance of the record type declared locally. Don't waste time debugging alignment errors.

You can make a union of all your record types, but you shouldn't need to.

Also, use the type codes, record structs, and any accessor macros or other material you have to teach dumpsfs to print out your journal records. You *will* be dumping the journal to see what's going on, probably a lot.

If you change kern/sfs.h, remember to do `make includes` in the top level of the kernel source to install the new version before rebuilding dumpsfs. (Rebuilding everything from the top does this but in general takes long enough to be annoying.)

We strongly recommend having a place (one specific place, maybe in a big comment in kern/sfs.h, maybe in the design doc) where you document exactly what the record types are, what each field of each record type is, and what each of these things means, and *keep it up to date* if/when you make changes.

Also, if you need to make substantive changes, either to the contents of a record or (more so) to its semantics, use a new record type name and new type code number and retire the old one. This

helps make sure all references in the code get updated, and also means that if you accidentally fail to and encounter the old record at recovery time you can panic instead of doing wrong things.

If you are really clear on what the records are and what they mean, it's perfectly feasible for one partner to work on the recovery code and one to work on the record issuing code. However, because it's difficult to test the recovery code until you have records coming out, this split might actually not work so well in practice.

4 The Container Code

Ideally you won't have to wade into `sfs_jphys.c`; but you do have to live with it, so here are some things you probably do want to know.

Verbose recovery. If you uncomment `SFS_VERBOSE_RECOVERY` in sfsprivate.h, the container-level recovery in `sfs_jphys_loadup` will print what it's doing. This can be useful. Even more useful is to use this hook to have your own recovery code print stuff while it runs. (If your experience is anything like mine, you'll leave verbose recovery on until you're almost done with the assignment.)

There's a `SAY` macro in sfsprivate.h that's a conditional wrapper around kprintf for printing stuff when verbose recovery is on. (There's also an `UNSAID` macro that you can use to avoid related unused variable warnings when verbose recovery is off.)

Internal records. The jphys code uses two record types of its own, pad records and trim records. Pad records appear when a journal block is not otherwise full. (However, if padding smaller than the size of a record header is needed, it's implicit.) Trim records keep track of where the on-disk journal tail is and are issued with `sfs_jphys_trim` during checkpointing. These records are not directly exposed (the journal iterators skip over them) but you will see them if you dump the journal, which you will do a lot.

The journal buffers. The jphys code keeps two buffers available: one for the current journal head (where new records are added) and one that's not directly used but will become the next head buffer. When the current head block fills up, that buffer's released and the next buffer is moved into place. This happens inside the journal lock.

However, it isn't safe to get more buffers inside the journal lock (because getting buffers can trigger buffer evictions, and evictions might need to flush the journal), so the new next buffer isn't fetched until after the current journal write is finished and the journal lock can be released.

If at this point other threads come along to write journal records, they block on a CV (`jp_nextcv`) until the new next buffer appears. This avoids cases where a flood of incoming requests can fill up the newly deployed journal head buffer before there's a new next buffer to replace it with. (If that happens, the jphys code panics.)

Flushing-related metadata. The jphys code keeps track of the first LSN in each journal buffer, and the oldest journal block that's still in memory. This material has its own spinlock (which you shouldn't have to interact with) and is used for flushing the journal out.

Head/tail collisions. The jphys code does *not* detect head/tail collisions. This is a bug, and will hopefully get fixed next year – if I have time it'll get fixed next week, but I can't make any promises.

If the in-memory head collides with the in-memory tail (meaning the whole journal is still in memory) you will probably deadlock. This is very unlikely to happen, because the syncer is making sure blocks get written out reasonably promptly and that should trigger journal flushes. If this happens, it almost certainly means you aren't flushing the journal.

If the on-disk head collides with the on-disk tail, the result will be that when you go to load up the journal it either can't find a trim record at all (if the last one disappeared under the tide of new journal records) or it finds a trim record but it can't find the journal tail. Apart from the cases with very large writes and truncates that we explicitly don't intend you to worry about, if this happens it almost certainly means you aren't checkpointing, either often enough or possibly at all.

5 Testing

There's a fairly wide variety of test workloads in testbin/.

Use the doom counter to crash the system in the middle. Early on, pressing Ĉ while the system's running will serve fine; later on, the advantage of the doom counter is that it will give repeatable results (or mostly repeatable results) and also, by incrementing the doom number one at a time you can exercise different combinations of buffers being written out.

Some of the tests have phases; crashing in the different phases is usually a materially different recovery problem. Particularly in frack, some tests have an explicit sync between phases; generally this means that the first part is setup for the second part and crashing in the second part is what's supposed to be interesting.

When you first start testing, don't even bother trying to run recovery; instead, just dump the journal and check it to see if it's correct and complete. (The first few times, it won't be.)

Once you're generating the right log records you can proceed to trying to recover with them. Testing recovery from wrong log records isn't very rewarding.

Poisondisk. The poisondisk program clears a disk image to a known nonzero value (0xa9). Run this on your test disk image, then run mksfs, then run a test workload; if you see 0xa9 coming through (or 0xa9a9, or 0xa9a9a9a9) you know you're getting stale block data.

Mostly this will manifest in user data if you aren't handling uninitialized user data right; but it can crop up in metadata structures too if you have the right (wrong) bugs.

`frack`. The frack (filesystem crack) test contains a large number of test workloads and a validator for them. As described in the assignment text, use `frack run` to run a workload; then after crashing and running recovery, use `frack check` to validate the filesystem state.

It will match what it finds in the filesystem against the sequence of states arising from the workload. If it doesn't find an exact match, it finds the closest match (the logic it uses for this is primitive and may not do a great job) and lists the discrepancies. When it finds zeroed ranges of files, it will announce them; this isn't "complaining" or a failure per se. If it finds the wrong data, it will complain.

Note that it doesn't react all that well to half-finished write operations; e.g. if the workload wrote out a 16K and 8K made it to disk (so the file length is 8K and the data is there) it will get

upset, whereas if it sees a 16K file with the same 8K of data and 8K of zeros from the unfinished write it will be fine with that. Since allowing writes to be half-finished is a reasonable way of dealing with partially written user data, it would be nice if frack were smarter about this; but it isn't entirely trivial to make it so.

6 Some Other Stuff

The freemap. As noted before/elsewhere, the free block bitmap is not handled through the buffer cache. This makes it a special case needing special handling. (This should really be fixed sometime.)

Treat it as a special case of buffer; give it the same metadata as a buffer and scan that metadata when you'd scan buffer metadata. Also, because by default it only gets written during an explicit sync, youll want to write it out from time to time.

Because `sfs_freemapio` *does* go through `sfs_writeblock` you will not need any special handling to ensure WAL for the freemap beyond scanning the freemap metadata just as you'd scan the buffer metadata to properly flush the journal in advance of writing blocks to disk.

Buffer reservation during recovery. As per a Piazza post over the weekend, before scanning the journal, reserve some buffers with `reserve_buffers`, and unreserve them again when done. Otherwise you'll assert inside the buffer cache. I think I lied last week about the need to do this.

7 The Issue in Truncate

I'm going to go through this because it's an example of the kinds of things that can go wrong at recovery time if one isn't careful.

The code in `sfs_itrunc`, as you may have noticed if you've waded into it, releases blocks with `sfs_bfree` as it goes, which in turn locks and unlocks the freemap for each change.

This means that blocks released by truncate can be reallocated for use by something else before the truncate completes. The in-memory state of things gets away with this because `sfs_itrunc` doesn't really attempt to unwind itself on error. However, in combination with journaling this isn't so innocuous.

Suppose that process P is truncating a large file. Now process Q creates a new file and calls `write` on it, which allocates blocks; let's say that the first data block Q gets for its new file is a block B released by P. The write finishes and commits, Q goes about its business, and the machine crashes before P's truncate finishes.

What happens then?

At recovery time, the write is a committed transaction so it isn't going to be undone. But the truncate is *not* a committed transaction, so it *is* going to be undone. Which means that block B is restored to P's large file; but it's also part of Q's new file. Now the volume's corrupt.

What was wrong that allowed this to happen?

- `sfs_itrunc` didn't follow the two-phase locking rule;
- Q's transaction read dirty data;
- this made Q's transaction *dependent* on P's, but it committed without waiting for P's transaction to commit first.

Implementing support for dependent transactions is possible but not what we want. The fix is to follow two-phase locking and not release the freemap lock until truncate is done; I'm working on a patch but need to come up with a solution that's noninvasive and that doesn't violate existing lock ordering constraints.