Assignment 4 Section Notes

CS161 Course Staff

April 2, 2016

1 Administrivia

- The in-class design review is on Tuesday, 4/12.
- Final designs are due Thursday 4/14 at 5:00PM.
- There is also a code drop with more system calls implemented for you. You'll need to do some merging to make them compatible with your file table implementation.
- Please pull and merge *before* tagging your tree for the start of assignment 4.

2 Assignment 4 Overview

Assignment 4 is to implement journaled recovery for SFS.

We give you container code (`sfs_jphys.c`) that manages an on-disk journal structure. It has interfaces for writing to the journal (as you generate journal records while running) and reading the journal (during recovery). This code is part of the 2014 solution set and has been tested and (mostly) debugged – but it's not inconceivable that it still has some issues. If in doubt, speak up.

You have two coding tasks:

- Write the code that issues journal records.
- Write the code that replays the journal and recovers the file system.

However, these have another prerequisite:

- Design a schema for the journal records you'll use.

This means: figure out what types of journal records you're going to have, and what they mean.

You really need to sort out the journal schema on paper before you write much or any code – not necessarily the exact final version, but as close to it as possible. It's important that by the time you start coding you are very clear on what journal records exist and exactly what they're supposed to mean, and if you have to make changes on the fly that you do so very carefully. (We'll talk more about this in a bit.)

Once you have the journal schema defined, you can write both the code that fills out the journal record and sends it to the on-disk journal, and the recovery-time code that, given an instance of that record found at recovery time and a direction (undo or redo) ascertains if the action needs to be applied or not, and if it does, apply it.

There are then some other pieces that need to be filled in as well:

- Enforcing the write-ahead property of the journal.
- Managing transactions.
- Generating periodic checkpoints and trimming the journal.
- Writing the recovery-time logic that scans the journal one or more times, reads records, and invokes the code that applies them.

We will cover all of these things in turn.

Note that you will be implementing record journaling (not block journaling) and you will be using low-level to mid-level records (physical records) and not high-level records (logical records). Block journaling is something quite different and not the subject of this assignment. Logical record journaling is tricky and leads to complicated recovery code. It's not wrong; we just believe it to be substantially more difficult and not likely to be a success in the time frame we have.

You will also most likely be implementing undo/redo logging. While redo-only or undo-only logging may seem simpler at first, as discussed in class they impose additional operational constraints at runtime that are a fairly substantial nuisance to implement, and not recommended.

Things you do not have to do:

- Worry about making sure the journal blocks get written to disk in order. The jphys code handles that for you; the necessary hooks are already in `sfs_writeblock()`.
- Figure out how to start up the journal at mount time, manipulate its operating modes, or create it in `mksfs`. This code is already in place. (You might conceivably want to change or add to what `mksfs` does; but this probably won't be necessary.)
- Add other file system features. SFS is already more or less complete, or at least as complete as it's going to get.

3 Transactions

You need to be able to group journal records into transactions. While it's theoretically possible to collect all the information you need to journal for a single operation into a single log record, and then just write that and not need to group anything, this both tends to make your journal records very high-level (which as noted elsewhere becomes tricky to get right and is not recommended) and turns out to be structurally difficult in the SFS codebase.

We talked about transactions in lecture; the basic idea is that a transaction is a collection of operations that either all happens or all doesn't happen. If you have seen/used transactions in databases, you'll find that journal transactions in a file system are a little different; if you haven't yet, keep this in mind for when you do.

In SFS we already have code that provides locking and maintains run-time consistency. The actions on the file system are organized into a family of specific operations, and in the absence of crashes, each of these operations is already atomic: either it happens, or if something goes wrong partway through any changes made so far get backed out and nobody else sees them. All our journal transactions need to do is make updates to the on-disk structures atomic with respect to crashing.

This makes a number of things simpler; for example, there is no need to create transactions for read-only operations like `lookup`. Only operations that modify the disk need to be transactions.

We'll talk more about how one handles transactions further on; for right now, it's sufficient to note that we need transactions and that they group small operations together into larger operations. Also recall from class that a transaction can either *commit*, in which case it takes effect, or *abort*, in which case everything it did is backed out and it is forgotten.

4 Designing Journal Records

4.1 Idempotence

As discussed in class, all records need to be idempotent: you need to be able to apply them over and over again without going off the rails. (Or, equivalently, you need to be able to tell whether they've been applied already so as to be able to consider them over and over again but still apply them exactly once.) "Increment x" is not idempotent; "set x to 3" is. In undo/redo logging since you want the journal records to be idempotent going either forward or backward, the typical formulation is old value/new value. However, that's not the only way to do it. (For example, if you store the LSN of the last update in the on-disk data structure, then you can get idempotence by comparing that to the LSN of the change record.)

4.2 Begin records

One of the things you need to decide is whether to have explicit transaction begin records or not. Strictly speaking there isn't any need for explicit begin records; you can set things up so that the first record that mentions a new transaction ID is implicitly a begin record, and this works fine. So having explicit begin records is a waste of space and disk bandwidth... except for one point: a begin record is a good place to dump out information about the transaction: what operation it is, what process is doing it, etc. This information makes reading log dumps *much* easier. Recommended.

4.3 Abort records

A commit record tells you that a transaction completed successfully. An abort record tells you that a transaction did not complete. Why might this not be all that useful at recovery time?

As another wrinkle, remember that SFS already contains code that undoes changes on error, as long as there isn't a crash first. This code needs to be run in order for the right things to happen at run time; therefore, when an operation fails and aborts, and all the changes get written out to disk before crashing, the net effect of the failed operation is nothing and there is nothing further to do at recovery time. Writing out an abort record at this stage does not add much value.

Finally, it turns out that provided you journal the updates made when cleaning up on error paths (which you should), there's no real difference between a commit record and an explicit abort record. In both cases the transaction is done and all the changes it's made can be enacted, because in the case of an abort the net changes it's made are nothing.

Given these circumstances, abort records aren't that useful. The chief value of explicit abort records is that they reduce the number of potentially live transactions you need to keep track of during recovery. They also (like explicit begin records) make reading log dumps easier.

4.4 Freemap issues

In the SFS code as it currently stands, the free block bitmap (or "freemap") is managed in memory with a struct bitmap and is not stored in the buffer cache. This causes operations on it to be special cases. In particular, you want to make sure that any record that is meant to affect the freemap is explicitly about the freemap and can be applied specifically to the freemap, and is in this respect logically different from records that affect other structures/areas of the disk.

Otherwise you end up with special cases in your recovery code to detect freemap changes and handle them differently, which is a major hassle. (I discovered this issue the hard way in the solution set.)

4.5 User data

You aren't required to journal user data. (In fact, because of the structure of the journal, it would be awkward to do so.) Therefore, it's ok to sometimes lose user data if it hasn't been written out yet, and because the buffer cache doesn't necessarily write out in order the contents of user files may not necessarily be consistent with what operations the user was doing right before the crash. (For example, if the user writes out new versions of blocks 1, 2, and 3 in that order, after a crash the user might find that blocks 1 and 3 but not 2 contain the new version.)

However, you *are* required to not expose uninitialized data in user files.

This set of circumstances creates a requirement: it must be possible to know, at recovery time, which blocks were newly allocated to a given file, and which of those blocks written out successfully and which didn't. Then you can zero the ones that didn't to avoid exposing the stale data that would otherwise be in them.

There are several ways to do this. One is to invert the write-ahead property for user data blocks: require that the contents of any new block be on disk before any of the journal operations that describe allocating the block and linking it into a file. This method is complicated, not performant (typically it means that any write operation that extends a file, which is most writes, becomes effectively synchronous), and not recommended.

The better alternative is some combination of logging checksums of user data blocks as they're written into the cache, so the checksums go in the journal along with the block allocation and linking and can be checked during recovery; and/or logging the fact that a user data block has been written once it finally gets written out from the buffer cache, so if you see that record in the journal during recovery you know the block itself is good.

Note that there's no need to zero a file data block if the version on disk is an older version that still belonged to the same file; that is, overwriting an old block can be treated differently from writing a new block. Drawing this distinction is good but under the ground rules of this assignment not strictly necessary.

It is also nice (but not necessary/required) to tag files that are corrupt after a crash.

4.6 User data 2

Not journaling user data creates a complication, however, which is that you can't either undo or redo changes to user data blocks. The version you find on disk at recovery time is what you get, and that's it.

This is not by itself a big deal. But it means that you have to be careful when user data blocks change to metadata blocks and vice versa. (For example, directory contents are metadata and changes should be journaled; but they're just blocks and can be freed and reused as user data blocks. In SFS inodes are just blocks too.) If you apply journaled metadata changes to blocks that are (as of crash time) user data, you'll irretrievably corrupt the user data.

We'll discuss exactly what this entails at recovery time when we talk more about recovery; from the perspective of designing journal records the key thing is that you need to be able to determine (easily) at recovery time when blocks change from metadata to user data and back by reading the journal. So e.g. you might want to explicitly include a metadata/user data flag in your records that describe block allocation and freeing.

4.7 Unlink vs. reclaim

In Unix it's possible to unlink a file and continue to use it; the file is not actually removed until it's closed. The close can happen arbitrarily long after the unlink, and the file can be arbitrarily rearranged in the meantime.

This means that unlink and reclaim have to be separate transactions, and that in turn means that in order to avoid losing track of files that have been unlinked but not reclaimed, you need to keep track of them explicitly on disk somehow.

This can be done purely by writing information to the journal (and rewriting it periodically to allow checkpointing to function) but it's generally easier to create a new on-disk structure to reference the unlinked files until they're reclaimed. Then you can just journal updates to this structure, and then after recovery go through and finish deleting anything that still needs reclaiming.

This is something you'll need to do. Use the program `/bin/tac`, which prints files backwards, to test this facility; it intentionally uses an unlinked temporary file so as to serve as a test case.

Note that the first thing reclaim does is truncate the target file; this might or might not be a separate transaction from freeing the inode.

4.8 Large writes

Very large write operations can generate very large transactions. One consequence of this is that a very large write can overflow the journal. You're explicitly not required to worry about this case. However, large transactions can interfere with orderly checkpointing and create other problems; and if one crashes halfway through a large write it's tempting to want to not throw away the file data that made it out before the crash.

Because we don't journal user data, it is possible to break large writes into a series of single-block writes. However, this can be somewhat tricky – if you attempt this be sure that there's no way to leave blocks allocated beyond end of file and that a failure partway through doesn't leave the file size in an invalid state.

Also keep in mind that because the journal is linear, there's nothing to be gained by issuing smaller transactions if you still need the beginning point of the combined large operation to recover successfully.

4.9 Truncate

Truncate can also generate very large transactions, and a very large truncate can also overflow the journal. You are also not required to worry about this case.

It is possible to subdivide truncates into smaller transactions, and it's tempting to try this. Many of the same issues as large writes still apply.

Also be advised that the truncate code in SFS is nasty and adding complicated logic to it will be painful.

5 SFS

In this section we'll quickly go through the SFS code and its structure.

5.1 VFS interface

SFS necessarily supports the abstract interface defined by the VFS layer. This consists of two sets of functions: operations on the whole volume, called `FSOP_*`, and operations on vnodes, called `VOP_*`. You've seen several of the latter before.

The fsops interface has these operations, defined as macros in fs.h:

- `FSOP_SYNC` - syncs the whole volume, that is, writes out all dirty data to disk. You may wish to make a checkpoint afterwards.
- `FSOP_GETVOLNAME` - this returns the volume name and is used by VFS-level code to allow addressing mounted volumes by name.
- `FSOP_GETROOT` - this returns a reference to the root directory vnode of the volume. Used by VFS-level code to translate device and volume names to vnodes.
- `FSOP_UNMOUNT` - unmount the volume cleanly. This can be triggered either by explicitly unmounting (from the kernel menu) or by shutting down without having unmounted first. Logic in the VFS layer calls `FSOP_SYNC` first after locking out new writers; so if the volume is idle such that unmount is allowed, code in unmount can assume the volume has been synced.
- `FSOP_READBLOCK` - read a block. This is what the buffer cache calls to read blocks. Anything that fetches buffers elsewhere in the SFS code will come back here when actual I/O is needed.
- `FSOP_WRITEBLOCK` - write a block. This is what the buffer cache calls to write blocks. All writes from the buffer cache come back here, whether they happen via an explicit buffer flush, via an explicit sync, or because the background syncer thread decides it's time the block in question got to disk.
- `FSOP_ATTACHBUF` - attach metadata to a block. When a new buffer is "attached" (we'll talk about what this specifically means to the buffer cache in a bit), it calls back into the file system to allow the file system to allocate and initialize per-volume metadata.
- `FSOP_DETACHBUF` - dispose of block metadata. This is the opposite of attachbuf and called when the buffer cache detaches (basically, discards) buffers. It is possible for a dirty buffer

to get here; e.g. if you truncate away blocks that have been modified, or reclaim and free an inode before earlier changes to it have been written out, those buffers will still be dirty when detached. However, if a dirty buffer appears here the data in it no longer has any value.

The SFS code for these operations, along with the mount logic, can be found in `sfs_fs.c`. The functions are called `sfs_sync`, `sfs_getvolname`, etc.

Exercise to see who's awake: why isn't mount a `FSOP`?

The vnode interface has these functions:

- `VOP_EACHOPEN` - called during open to allow rejecting unwanted open modes. Read-only.
- `VOP_RECLAIM` - called when the reference count of a vnode is about to decrease to 0. This can modify the file system (it frees inodes) and will need journaling.
- `VOP_READ` - called to read file data. Read-only.
- `VOP_READLINK` - for reading symbolic links; read-only and not implemented by default.
- `VOP_GETDIRENTRY` - for reading directories; read-only.
- `VOP_WRITE` - called to write file data. Will need journaling.
- `VOP_IOCTL` - for miscellaneous actions; SFS doesn't have any, so doesn't need attention.
- `VOP_STAT` - retrieve file information. Read-only.
- `VOP_GETTYPE` - retrieve file type. Read-only.
- `VOP_ISSEEKABLE` - check if an object is seekable. Read-only.
- `VOP_FSYNC` - flush file data. Probably doesn't need explicit attention itself.
- `VOP_MMAP` - map a file into memory. Provided only as a hook by default and not implemented.
- `VOP_TRUNCATE` - change file size. Will need journaling.
- `VOP_NAMEFILE` - implements getcwd; read-only.
- `VOP_CREAT` - create a new file. Will need journaling.
- `VOP_SYMLINK` - create a symbolic link. Not implemented by default.
- `VOP_MKDIR` - create a directory. Will need journaling.
- `VOP_LINK` - link a file into a directory. Will need journaling.
- `VOP_RMDIR` - remove a directory. Will need journaling.
- `VOP_RENAME` - rename a file or directory, including across directories. Will need journaling.
- `VOP_LOOKUP`, `VOP_LOOKPARENT` - translate a pathname to a vnode. Read-only.

Most of these functions are in `sfs_vnops.c`. The functions are called `sfs_read`, `sfs_write`, etc.

Note that `sfs_reclaim` is in `sfs_inode.c` along with other vnode/inode lifecycle functions, and the guts of truncate are in `sfs_bmap.c` with other indirect-block-traversal code.

5.2 SFS structure

The SFS code is split into 8 modules, as follows:

- `sfs_fsops.c` - whole-volume operations, including mount/unmount
- `sfs_vnops.c` - vnode operations
- `sfs_dir.c` - directory operations
- `sfs_bmap.c` - block lookup within files; inode traversal
- `sfs_inode.c` - lower-level inode operations; vnode/inode lifecycle code
- `sfs_balloc.c` - block allocation
- `sfs_io.c` - block-level I/O routines
- `sfs_jphys.c` - the code for the on-disk journal

The declarations exposed to userlevel (and used by mksfs, dumpsfs, sfsck, and any other tools that might appear) are found in `<kern/sfs.h>`.

The declarations exposed to the rest of the kernel, but not userlevel, such as `struct sfs_vnode` and `struct sfs_fs`, are found in `<sfs.h>`.

Internal declarations are found in `sfsprivate.h` within the SFS source directory.

5.3 Important internal functions

In `sfs_dir.c`:

- `sfs_readdir` - fetch a directory entry from a directory
- `sfs_writedir` - update a directory entry in a directory
- `sfs_dir_findname` - find a directory entry by name
- `sfs_dir_findino` - find a directory entry by name
- `sfs_dir_link` - insert an inode into a directory
- `sfs_dir_unlink` - remove an entry from a directory

In `sfs_bmap.c`:

- `sfs_bmap` - translate file blocks to disk blocks
- `sfs_itrunc` - guts of truncate

In `sfs_inode.c`:

- `sfs_dinode_load` - load an on-disk inode into memory
- `sfs_reclaim` - `VOP_RECLAIM` implementation

- `sfs_loadvnode` - find or load a vnode by inode number
- `sfs_makeobj` - create a new file/directory

In `sfs_balloc.c`:

- `sfs_balloc` - allocate a block
- `sfs_bfree` - free a block

In `sfs_io.c`:

- `sfs_readblock` - low-level block read
- `sfs_writeblock` - low-level block write
- `sfs_io` - backend for user-data bead and write operations
- `sfs_metaio` - backend for metadata read and write operations

5.4 Buffer cache

The buffer cache code is in `kern/vfs/buf.c` and the interface is in `<buf.h>`.

Rather than going through the API (which is documented in buf.h) we'll go over things you need to know about the buffer cache and its terminology.

First, the buffer cache is physically indexed. Buffers are looked up by file system (`struct fs`) and disk block number. (It is also possible for a buffer cache to be virtually indexed, so lookups happen by file block number before file block → disk block translation in `sfs_bmap`. There are various possible reasons for that; but we don't do it.)

Attached buffers. A buffer is "attached" if it has a file system and disk block number that it's a buffer for. Detached buffers are free. It's important that attaching a buffer is atomic; otherwise concurrent requests could create two buffers for the same block and things would deteriorate fairly rapidly afterwards.

Busy buffers. A buffer is "busy" while a file system is using it. This is also equivalent to "locked". A busy buffer is not written out and cannot be evicted until released. Retrieving a buffer (with `buffer_get` or `buffer_read` returns a busy buffer. File system code should never see a non-busy buffer. Buffers are unbusied by releasing them, with `buffer_release` or `buffer_release_and_invalidate`.

Valid buffers. A buffer is "valid" if it contains data. (An invalid buffer still contains data, obviously, but the data it contains is trash.) The difference between `buffer_get` and `buffer_read` is that if the block requested isn't already in memory, the former will return a non-valid buffer (that should be filled with data and marked valid) and the latter will go to disk to read the buffer in.

The difference between `buffer_release` and `buffer_release_and_invalidate` is that the latter clears the valid bit on the buffer.

Dirty buffers. A buffer is, unsurprisingly, dirty if its contents haven't been written to disk since they were last changed. File system code marks buffers dirty; they get unmarked internally after being written out. Invalid buffers are never dirty.

Per-volume metadata. In order to help support write-ahead logging and other file system logic, buffers have a hook for per-volume metadata. When a buffer is attached, the buffer cache

calls FSOP_ATTACHBUF, and if desired the file system code can therein use buffer_set_fsdata to change the per-volume metadata pointer. (This call returns the old metadata pointer, which in attachbuf should always be null.) Similarly, when a buffer is detached, the buffer cache calls FSOP_DETACHBUF, and the file system code should use buffer_set_fsdata to set the per-volume metadata pointer to null and destroy the old metadata that the call returns.

The existing sfs_attachbuf and sfs_detachbuf functions make the buffer_set_fsdata calls (to show how it's done) but don't actually install any metadata. You'll be changing that.

"fsmanaged" buffers. Buffers that are "fsmanaged" are not held by a specific thread (ordinary buffers are) and are exempt from the normal buffer reservation logic. These buffers are ignored during the syncer and sync routines (e.g. buffer_sync and sync_fs_buffers); they must be explicitly flushed by the file system. The buffers used inside sfs_jphys.c to hold the journal are "fsmanaged"; you shouldn't see or need others.

Buffer reservations. On entry, file system operations that use buffers (which is most of them) call reserve_buffers to reserve some buffers for use. The number of buffers reserved is a constant in buf.c that's supposed to be enough for any one operation - if you find that you need more somewhere, bump that number. Trying to get a buffer without first reserving some is an error and will assert. If there aren't enough buffers, trying to reserve some will block; the purpose is to avoid deadlocks where e.g. every thread is in the middle of an operation, has two buffers, and needs one more, and there aren't any left.

The buffer lock. There's one (sleep) lock for the buffer cache. It is used to protect most buffer metadata (some is protected by the buffer being busy) and is released during I/O and on all calls back into the file system. The buffer lock is not exposed to file system code (only locks on individual buffers are) and does not impose lock ordering requirements on it.

Internal structures. There are four primary internal tables in the buffer cache: there's a hash table for looking up buffers by file system and block number, an LRU queue of attached buffers, a separate queue of dirty buffers, and finally an array of detached buffers where free buffers sit.

Epochs. The buffer cache keeps a global epoch, which is basically a counter of the number of explicit syncs that have been done. Each explicit sync call (sync_fs_buffers) writes out only buffers that became dirty in the same epoch; this keeps sync from running forever if called while a volume is busy.

Syncer. There's a background syncer thread. It wakes up once a second and looks for work to do. It is supposed to keep buffers from remaining dirty for more than two seconds at a time, to reduce data loss in crashes. The syncer uses substantially different code paths from explicit sync calls because it's specifically intended to not just write everything out every time it runs.

Buffer allocation. Buffer space is allocated with kmalloc as it is first used, up to a limit based on the size of RAM. Once allocated, buffers are never freed (just reused) - the design of this logic was motivated by the desire to avoid having the buffer cache using memory during assignment 3.

6 The Container Interface

Like with the buffer cache, rather than going through the interface specifically (which is documented in design/jphys.txt we'll go over the concepts and terminology associated with the jphys code.

Heads and tails. The on-disk journal is a circular buffer. The "head" of the journal is where new records are written. The "tail" is the oldest record that's still needed for recovery.

Log sequence numbers. A log sequence number, or LSN, identifies a specific log record. LSNs are unique, nondecreasing, and assigned inside `sfs_jphys_write` when you send records to the journal.

Internal records. The jphys code has some internal record types that it uses to implement pieces of its own functionality. These are not exposed to higher-level code. We'll talk more about them next week.

Modes. The jphys code has separate reader and writer modes that need to be explicitly turned on and off. This helps avoid mistakes with locking and with its internal initialization. The code that manipulates these has been written for you and you probably don't need to change it.

Note that reading from the log during operation is not intended to be supported, isn't really safe, and probably won't do what you want. Writing to the log during recovery is allowed if enabled (provided that you don't write so much as to overwrite the original log tail, at which point the behavior is undefined) but not recommended.

Iteration. The jphys reading interface is used to scan the journal during recovery. It consists of an abstract type `sfs_jiter`, a journal iterator or cursor, and typical cursor-type operations on it. You can scan in either direction. (You can even use the same cursor to go both directions, with some caveats.) Only the range of the journal between the head and tail is iterated. The jphys internal records are automatically skipped over.

The iteration model is:

```
sfs_jiter *ji;
unsigned type;
sfs_lsn_t lsn;
void *ptr;
size_t len;

sfs_jiter_fwdcreate(sfs, &ji);  /* or _revcreate */
while (!sfs_jiter_done(ji)) {
   type = sfs_jiter_type(ji);   /* record type code */
   lsn = sfs_jiter_lsn(ji);     /* LSN of this record */
   ptr = sfs_jiter_rec(ji, &len); /* data in this record */
     ...
   sfs_jiter_next(sfs, ji);     /* or _prev */
}
sfs_jiter_destroy(ji);
```

Note that `sfs_jiter_next` and `sfs_jiter_prev` can fail, e.g. on I/O errors. (This is why they're separate from `sfs_jiter_done`.) The creation functions can of course fail as well, as they need to allocate memory.

Issuing log records. To write a record into the journal, use the function `sfs_jphys_write`. It takes a type code (you can use any values in the range 0-127) and a blob of record data, and returns an LSN. There's also a callback function... more on that later.

Flushing the journal. In order to implement write-ahead logging, you will at times need to flush the journal, either up to (and including) a particular LSN, or all of it. The functions `sfs_jphys_flush` and `sfs_jphys_flushall` do this. Two additional functions are used for making sure the journal gets written in order; the hooks for this are already installed in `sfs_writeblock` and should not need further attention.

Checkpointing. To support checkpointing there's a function `sfs_jphys_trim` that discards an old section of the log. The LSN passed to this function should be the oldest LSN that is still required to recover the volume. You will find/compute this LSN (more on that later) – the function `sfs_jphys_peeknextlsn` is provided to give the LSN to use if there are no older LSNs required and the journal can be emptied. Note that, like with many other functions of this type, more log records can potentially be added between the call and the time you interpret the results; so the value used should be treated as a conservative approximation for checkpointing purposes and is otherwise mostly unsafe.

There is also a "block odometer" that tells you how many journal blocks have been used since the last time it was cleared. This can be used to schedule checkpoints if desired.

7 Locking

The jphys code has two locks: the main journal lock (`jp_lock`), which protects the journal state, the journal head position, and so forth. This lock is held while adding records to the journal. There is also a separate spinlock, `jp_lsnmaplock`, that protects certain data used to flush the journal.

The journal lock comes after other SFS locks, because in general you need to be able to add records to the journal while you have arbitrary other data structures locked.

It is not safe to hold the journal lock while getting buffers, because getting a buffer can trigger a buffer eviction and that can trigger flushing the journal; also in some designs that can sometimes trigger writing further journal records. Therefore there's an additional CV inside the journal (`jp_nextcv`) that one sometimes waits on when a new buffer is being fetched for the journal head. (This is an implementation issue, so more on it next week.)

8 Issuing Journal Records

In general, calls to `buffer_mark_dirty` are places you need to issue journal records. You will need to make your current transaction information available at these points; this can be done by passing it around or by sticking it in the process or thread structure.

In general you want to set up your journal records so that the information you need to journal is immediately available at the point you need to journal it. Sometimes this means issuing more journal records than strictly necessary for some operations; that's fine.

9 Enforcing Write-Ahead

As discussed in class, the cardinal rule is that no data or metadata buffer can go to disk until all the journal records describing changes made to it are on disk first.

This means two things. First (and most basic), you need to issue journal records before releasing buffers. Since buffers aren't written while they're busy, it isn't strictly necessary to issue journal records before changing the buffer contents themselves; but it's usually a good idea.

Second, you need to insert code that makes sure the journal gets adequately flushed before any non-journal block gets written. This requires adding a hook to `sfs_writeblock` that figures out what LSN must be written out and calls `sfs_jphys_flush`.

In order for that hook to work, you must keep track, for each buffer, of what LSN that is, and keep the information updated as you modify buffers and issue journal records. We recommend you use the provided per-volume metadata hooks in the buffer cache for this, although it's also workable to create an entirely separate lookup table.

The rule is: for a buffer b, the LSN that you must flush to (up to and including) is the LSN of the most recent record associated with a change to b. (If the buffer isn't dirty, then there is no such LSN.)

Something that's easy to forget: the block freemap, which is kept in memory and not accessed via the buffer cache, is a special case of buffer and also needs write-ahead journaling enforcement.

10 Checkpointing

We have apparently failed to talk about checkpoints in lecture, at least as of when these notes were written. So we'll begin with an overview before moving on to design considerations and requirements.

10.1 Overview

If you just write log records, eventually your journal fills up. At some point you have to flush out old chunks of the journal that no longer matter any more so you can reuse the space.

The procedure for doing this is known as *checkpointing*. A checkpoint is a record, or group of records, specifying that everything in the journal up to some point is no longer needed because all the changes involved have successfully been written to disk. Checkpoints also often include summary information or other material needed or desired during recovery.

There are two basic kinds of checkpoints: the conceptually simple kind, known as "stop the world", and the operationally simple kind, known as "rolling checkpoints".

For stop-the-world checkpoints you lock out all new writers and wait for all current writers to finish, so that no writes are happening. Then you sync all data, and write out a checkpoint that says "this is where the journal begins".

If after that point you crash, nothing earlier in the journal matters, and you recover by finding the most recent checkpoint and rolling forward from there.

(Let's make sure everyone understands this before going on.)

Stop-the-world checkpoints are a major nuisance to implement, and they're a performance bottleneck too. So instead, what everyone does (and what you will do, unless you like creating work for yourself) is what's known as *rolling checkpoints*.

The idea of a rolling checkpoint is that you don't bother to stop anything; just from time to time you go through and figure out what's made it to disk so far and what hasn't, and what LSN that corresponds to. Then you write out a checkpoint that says "the journal begins at LSN k", where typically k is some distance behind where the checkpoint itself is. So at no time is the journal necessarily empty, unless the volume becomes idle of its own accord and you happen to take a checkpoint during that time. But you're still purging old journal records once they aren't needed any more.

10.2 Checkpointing with jphys

The jphys code (as mentioned already) provides a function `sfs_jphys_trim` that trims old records off the end of the journal. This writes out an internal jphys-level record that tells the jphys code where the tail of the log is, and the record named in the last trim is the bound used by the journal iteration code.

You might or might not need or want to issue other records as part of a checkpoint; if so, they should go out *before* the trim record and the LSN you pass to `sfs_jphys_trim` should be computed to avoid discarding them. (If you write checkpoint metadata after the trim record, it might not make it out if you crash right then; in that case if the trim makes it out, the metadata from the previous checkpoint will be discarded and you lose. This is an instance of the write-ahead logging rule: the checkpoint metadata constitutes journal records describing the trim, so needs to go out first.)

You can implement either stop-the-world or rolling checkpoints with `sfs_jphys_trim`, but rolling checkpoints are a no-brainer choice.

10.3 Rolling checkpoints

In order to issue a rolling checkpoint you need to know (or be able to discover) the oldest LSN in the journal that you still need for recovery.

This means two things: you must not throw away part of any currently active (unfinished) transaction, for what are hopefully obvious reasons, so you need to know the LSN of every transaction's first record. And, for every dirty buffer, you must not throw away any of the journal records for the changes made to it, in case you need to reapply those changes after crashing. So for every dirty buffer you need to know the oldest LSN that modified it. (Recall that for dirty buffers you also need to track the *newest* LSN that modified it. It's neatly symmetrical.)

It is also sufficient for buffers to keep the LSN of the first record of the oldest transaction that modified it, which may be easier to wrangle depending on design details. (Or you can even just refer to that transaction, although that creates other complications managing transactions.)

When it's time to checkpoint, you scan all your transactions and all your dirty buffers to find the oldest LSN still required, and trim to that LSN. (If there are none, you can call `sfs_jphys_peeknextlsn` to trim to an "empty" journal containing just the trim record; or if you have checkpoint metadata, to the LSN of the first checkpoint metadata record. There's two traps here, one minor and one major. We'll get back to those shortly.)

Clearly the buffer metadata and transaction information needs locking, and you need to be able to iterate over all dirty buffers (or maybe just all buffers) and all live transactions. This is something you need to design. Ideally you can arrange to iterate over your buffer metadata structures without needing to change buf.c or lock buffers directly.

10.4 The minor trap

In general you want to call `sfs_jphys_peeknextlsn`, or issue checkpoint metadata, *before* scanning the dirty buffers and transaction lists. Otherwise, unless that entire scan is atomic (which might be awkward or expensive) someone could add a transaction or a dirty buffer after you scan but before you peek at the next LSN, and then the next LSN you see will be after whatever they did and you'll accidentally throw away any records they added.

Also note that peeking at the next LSN necessarily takes the journal lock, so if you peek at the next LSN while holding locks associated with scanning this creates an ordering constraint between those locks and the journal lock that might be problematic.

Remember that once you call `sfs_jphys_peeknextlsn`, it doesn't matter how much more work happens or records get added before you trim – trimming to that LSN won't discard those newer records.

The major trap is similar but a topic in its own right, so we'll get back to it in a bit.

11 Managing Transactions

As we just described, in order to checkpoint you need to keep a list or table of all active transactions, and for each one you need to know the LSN of its begin record, or its first record if you don't have explicit begin records. This list or table needs to be locked somehow.

You also need, as we mentioned a while back, to make at least the transaction ID available when issuing journal records, which can be done by passing around the current transaction or sticking it in the process or thread structure. The latter assumes you don't nest transactions or otherwise have more than one per process at a time; but that's probably a reasonable assumption.

The remaining question is how to identify transactions. There are three basic options:

- Use the LSN of the begin record. Simple, easy, but LSNs are 64 bits wide and this wastes space. (Which doesn't really matter, but it grates.)
- Use a separate global counter to assign them. Also simple and easy, but creates a bit more bookkeeping.
- Use the process ID of the process issuing the transaction. Works fine if you don't have multithreaded processes (if so you need the thread ID too) but creates a slight added complication at recovery time, which is that the "same" transaction (by ID) can appear many times in the journal.

Any of these is a fine choice.

Tip: when passing around transactions (or even if stuffing them in the process structure), even if you think all you need is the transaction ID, use a pointer to your transaction structure. Then if you discover you need something in addition to the transaction ID, you don't have to make wholesale menial changes everywhere.

12 The Major Trap (or, that write callback)

You may have noticed that `sfs_jphys_write`, the interface for writing journal records, takes a callback function, and this might seem a bit scary.

It's there to allow dealing with an otherwise very awkward race condition.

12.1 The scenario

Suppose you're starting a new transaction, and there are currently no dirty buffers and no active transactions. You write a begin record (or the first record, if you don't have explicit begins) into the journal. This locks the journal lock, assigns an LSN, unlocks the journal, and returns it to you. Let's call this LSN value b.

Now suppose at this point the checkpointer runs. It calls `sfs_jphys_peeknextlsn` to get a fallback LSN for trimming. This locks the journal lock, checks the next LSN, unlocks the journal, and returns the next LSN k, where $k = b + 1$. Now it checks through the list of active transactions (none), checks all dirty buffers (none), concludes there's nothing needed in the journal, and trims the journal to k.

Now the scheduler gets back to running your new transaction. It gets whatever lock(s) it needs to update the active transactions table, and adds the new transaction. This creates a constraint on the checkpointer that the first record of this transaction, b, is still needed.

Unfortunately, the checkpoint that just ran already threw away your begin record at b by trimming to $b + 1$.

12.2 What happened

The problem is that once the journal's unlocked, arbitrary other stuff can happen; but the new transaction isn't reflected in the active transactions table until that's been locked and added to.

In between there's a window where an active transaction really exists but it can't be seen by the checkpointer. (Or by anyone else.) Boooo.

12.3 The fix

The fix is to update the list of active transactions to add the new one *before* the journal is unlocked. Generally it's not safe to lock the list of active transactions while calling `sfs_jphys_write`, as that call can get buffers and trigger buffer evictions and writes. (Your design may vary though.)

So the callback argument to `sfs_jphys_write` makes it possible to do this: it is called with the new LSN and a context structure (you can define it however you want) before the journal lock is released. Then the callback can lock and update the active transactions list, and unlock it again before returning. Note that this means the lock or locks protecting your list of active transactions must come *after* the journal lock.

(If you don't need to do this, e.g. because you're adding a record to an already existing transaction, just pass `NULL`.)

Yes, this is messy... if you come up with a better way of handling this situation please share it.

13 Recovery: Scanning

The basic procedure for recovery is to do two journal scans, one redo and one undo. For undo you scan backward, and redo forward, so the operations get (un)applied in the right order.

You can do the undo and redo scan in either order. That is, either you scan forward redoing all records, then scan backwards undoing records from transactions that didn't complete; or you scan backward undoing all records, then scan forward redoing records from transactions that *did* complete.

In the abstract ideal journaling database land, that would be sufficient. In our world because we don't journal user data, and we need to zero out uninitialized blocks, and because we have to look out for blocks that change from user data to metadata and vice versa, you'll probably need at least one more scan. The solution set does four total, the basic two plus one to find block type transitions and one to deal with stale blocks, but this shouldn't be considered authoritative.

You need a scheme to avoid overwriting user data in blocks that change from metadata to user data. In addition to making sure your records support this functionality (as we already mentioned) you need a plan for what to do at recovery time.

Similarly, you need a scheme for figuring out which blocks are uninitialized data in user files and zeroing them or otherwise avoiding exposing them, both in your journal records and at recovery time.

There's a general principle about recovery, which is: at recovery time you want, as much as possible, to be able to tell exactly what you have to do by looking directly at journal records. The less information you have to accumulate and track while scanning the better. Conversely, you do not want to get into the position of having to deduce or infer things during recovery. The more complex your recovery code is, the less likely it is that it'll work correctly all the time.

14 Recovery: Applying Records

For every record you have, you should have one function that applies it (redoes it) and one that unapplies it (undoes it). Call these functions when scanning.

That's about it, except for one thing about state: provided the information in the journal is correct, and your records are idempotent, and ignoring for the moment the issue of blocks that become user data, and because you're replaying in order, applying the sequence of changes in the journal to a particular thing is guaranteed to eventually reach the same end state that the journal describes. This is true even if you pass through some apparently incorrect states by applying earlier changes to a later on-disk state.

E.g. if you find the link count on disk is 2, and your journal records about it say

- change link count from 4 to 3
- change link count from 3 to 2
- change link count from 2 to 3
- change link count from 3 to 5
- change link count from 5 to 1
- change link count from 1 to 2
- change link count from 2 to 1

even if the rest of the inode (or even the whole disk) is in a state corresponding to between the last two lines, if you apply the changes starting from the 3rd one, skipping the first two because they don't match, you still come out in the right end state (1). (This is also true if the records aren't meant to support undo and contain only the new value and you blindly apply all of them.)

It is (I think) possible to break this principle by being inconsistent about the granularity of the journal records, which is another reason that high-level logical record logging is hard.

15 Recovery: After Scanning

Typically one doesn't write to the journal while recovering and applying (or unapplying) changes from the journal, because all the updates to the filesystem state being made have already been written to the journal and don't need to be written again.

However, once that's done there's generally some other work to do.

First, after recovering typically one checkpoints, to avoid having to do the same recovery work again.

Then one does any actions implied by the recovered volume state that may require issuing new journal records. In our case the important thing in this category is to go through the on-disk records of files that were unlinked but not reclaimed, and for each do the reclaim, which possibly includes truncating and may generate more than trivial amounts of log traffic. After doing that it's probably a good idea to checkpoint again.

16 Testing

Take note of what the assignment text says about the doom counter, the `frack` test, and the `poisondisk` tool. We'll talk about them next week as these notes are long enough already.