Introduction

This design doc is organized into the following sections:

1. Overview
2. Journal Entries
3. Recovery
4. Logged Operations
5. WAL Enforcement & Buffer Metadata
6. Transactions
7. Checkpointing
8. Graveyard
9. Testing

Overview

Our design contains a series of operation-level log records that help us recover after a crash. Each log record is associated with a corresponding redo/undo routine, a transaction, and before/after information to ensure idempotence.

Each system call that modifies the disk data (read-only ones are not journaled) creates a transaction. A transaction holds an identifier, which in our case is the process's pid. Several log records may be associated with one transaction, and each log record belongs to exactly one transaction. We have explicit trans_begin and trans_commit log records to determine where a transaction starts and whether it finished successfully or not.

To ensure we do not run out of journal space, we checkpoint every time the journal's odometer passes a certain point. In the checkpoint, we go through the journal and remove all unneeded transactions (more in 5. Transactions).

During recovery, we make three runs through the journal. One backward run finds potential garbage blocks. A second forward run builds up a list of uncommitted transactions, and it additionally tracks untouchable user data and additional potential garbage information. We finally do a forward pass through the journal, redoing all committed operations while respecting user data. Once the directory structure is restored, we checkpoint and try to delete all "dead" files that are no longer linked. If we finish recovery, we run a checkpoint and clear space for the new journal. If we crash during recovery, the journal will remain the same, and we can go through the same process on the next boot.

Journal Entries

The following journal entries with their component fields were added in our implementation:

`BLOCK_ALLOC`: Created whenever a block is allocated.
`BLOCK_DEALLOC`: Created when a block is deallocated.
`BLOCK_WRITE`: Created when a block is written to (either partially or fully)
`INODE_LINK`: Created when the link count of any inode changes
`INODE_UPDATE_TYPE`: Created when the the type of an inode changes

`META_UPDATE`: General purpose data updated. Created when directory entries are changed or on-disk block pointers change (e.g. in an inode)
`RESIZE`: Created when the size of an inode changes.
`TRANS_BEGIN`: Created at the start of a transaction.
`TRANS_COMMIT`: Created when a transaction completes successfully

Every journal entry contains the following information (with some exceptions):

`code`: An `int` representing the type of the transaction.

`id`: An `int` holding the id of the transaction that this entry belongs to.

`disk_addr`: A disk address for the block that was modified by the operation the journal entry is logging. (Not present in `TRANS_BEGIN` and `TRANS_END`)

These members are always stored in this order in each struct, and are the first three members of each struct. This allows us cast any journal entry to an int array and access these members when it is not necessary for us to know the type/code of the journal entry.

The journal entries have additional data associated with them so that we know the location of the data that changed, the original value of the data, and the new value of the data. For each journal entry, they are as follows

`BLOCK_ALLOC`: No additional content. *N.B. `ref_addr` and `offset_addr` are unused and always 0.

`BLOCK_DEALLOC`: No additional content

`BLOCK_WRITE`
`new_checksum`: Checksum of the data to be written (using Adler-32)
`new_alloc`: Null on creation, True in recovery if this followed an allocation
`last_write`: Null on creation, True in recovery if no other writes to this block

`INODE_LINK`
`old_linkcount`: The number of times that the inode is currently linked to
`new_linkcount`: The updated number of times the inode is linked to

`INODE_UPDATE_TYPE`
`old_type`
`new_type`

`META_UPDATE`
`offset_addr`: The offset within a block where the data was changed
`data_len`: The number of bytes that were changed

`RESIZE`
`old_size`: Original size of the inode
`new_size`: New size of the inode

`TRANS_BEGIN`
`trans_type`: Type of transaction

`TRANS_COMMIT`
`trans_type`: Type of transaction

Because the `META_UPDATE` journal entry has variable size, the old and new data documenting the change are not stored in the struct itself. The size of the journal entry for `META_UPDATE` is set at `sizeof(struct meta_update) + 2 * meta_update->data_len`. The old metadata is then copied immediately after the actual `META_UPDATE` struct, and the new metadata is copied immediately after that.

Whenever a journal entry is written, we also look up the buffer that was modified in the operation that the journal entry tracks. We will set the metadata of that buffer to track the lsn of this journal entry as the most recent to modify that buffer (for checkpointing purposes), and to track the lsn of the first operation to modify that buffer after it was written and marked clean. (See the Buffer Metadata section)

Recovery

General

During recovery, we make three runs through the journal:

1. Backward run: For any written block, find the last `BLOCK_WRITE` operation to touch each block. These operations will need to be checksummed.
2. Forward run: Build up a list of uncommitted transactions and blocks that ultimately turn into user data
3. Forward run: Redo all committed operations while respecting user data

Once the directory structure is restored, we checkpoint and try to delete all “dead” files that are no longer linked (see 7. Graveyard). If we finish recovery, we run a checkpoint and clear space for the new journal. If we crash during recovery, the journal will remain the same, and we can go through the same process on the next boot.

1. Backward run

We create a bitmap to track which blocks have already been processed. Whenever we see a `BLOCK_WRITE`, we check what block it references. If that block is marked as “processed” in the bitmap, we can ignore this block, since being “processed” means that there will be another write to the same block — we will have already seen such a write since we are iterating over the journal backwards. If the block is not marked as “processed,” then we mark that block as “processed” and set the `last_write` member of the `BLOCK_WRITE` struct to `true`. The “processed” flag tells us that any writes that we see henceforth are irrelevant since their data will be overwritten by this last write.

2. Forward run

We create two bitmaps. One bitmap tracks which blocks are userdata at the end of all operations. When we see a `BLOCK_WRITE`, we mark the disk address. When we see a `BLOCK_DEALLOC`, we unmark it. After processing all journal records, the bits set in the bitmap correspond exactly to those blocks ended up as user data. When recovering each journaled operation, we ignore those

that touch blocks that have their bit in the bitmap set, i.e. we ignore any operations that touch user data blocks.

The second bitmap tracks which blocks have been newly allocated. When we see a `BLOCK_ALLOC`, we mark the referenced disk address. When we see a `BLOCK_WRITE`, we unmark the referenced disk address and set the `new_alloced` member of the `BLOCK_WRITE` struct to true. When recovering these operations, we can tell if this write was to a freshly allocated block that may contain garbage data. If this write failed, then this block needs to be zeroed.

We also create an array of transaction structs. These transaction structs contain an array of operations that were part of that transaction, a bool to track whether or not that transaction was committed, and an id to look up the transaction for each operation. The transaction struct array tracks the "active" transactions at each step in recovery. These are the transactions for which we have not yet seen a "commit" record. At the end of this forward run, the elements of the active transaction array are exactly those that do not have a commit records. We treat these as aborted, and eventually undo all records that are part of these transactions.

3. Recovery (forward run)

We finally iterate once more over the journal. For every journal entry, we check to see if that operation was part of an aborted transaction. If not, then we redo this operation.

Checksums

We used the Adler-32 checksum pulled from Wikipedia to compute checksums.

Per Journal Entry

The `sfs_recover_operation` function in `sfs_fsops.c` is responsible for redoing or undoing the operation for a specific journal entry.

When recovering a specific journal entry, we first check if the referenced block is userdata. If it is, we short-circuit and return without doing anything, unless it is a write, in which case we may need to check it (see `BLOCK_WRITE` recovery). If the referenced block is not user data, we read in the block that that journal entry references, except for those that do not modify a block. We can then cast this block to a dinode or any other type of block.

BLOCK_ALLOC

Redo:

- `bzero` a buffer and write that to the block that was allocated
- If the freemap bit at this disk address is unmarked, mark it

Undo:

- If the freemap bit at this disk address is marked, unmark it

BLOCK_DEALLOC

Redo:

- If the freemap bit at this disk address is marked, unmark it

Undo:

- If the freemap bit at this disk address is unmarked, mark it

INODE_LINK

Redo:

- If the current `linkcount` of the inode is the journalled `old_linkcount`, set the `linkcount` to the journalled `new_linkcount`. Otherwise do nothing.

Undo:

- If the current `linkcount` of the inode is the journalled `new_linkcount`, set the `linkcount` to the journalled `old_linkcount`. Otherwise do nothing.

In both cases, write the inode back to disk

META_UPDATE

The `data_len` member of the `META_UPDATE` struct allows us to know how much data needs to be copied into the existing buffer. The old data is located immediately after the end of the struct; the new data is located immediately after the old data.

Redo:

- copy the new data over the current data at the location specified by the offset

Undo:

- copy the old data over the current data at the proper location

Again, we force this operation to write the block to disk

RESIZE

Redo:

- If the current `sfi_size` is the journalled old size, set it to the new size. Otherwise do nothing.

Undo:

- If the current `sfi_size` is the journalled new size, set it to the new size. Otherwise do nothing.

Write this inode back to disk

BLOCK_WRITE

Redo:

- If the write operation is the last write to touch the block, i.e. the journal entry has `last_write` set, then this write operation was the operation that created the user data

that is currently on disk. In this case, we need to check that this write made it to disk successfully (we don't need to check others since they are overwritten by this write).
- If the checksum of the block on disk does not match the checksum of our journal record, then we know that this write failed.
- If this write failed and the block was also newly allocated (`new_alloc == true`), then we know that this block contains garbage data. In this case we fill it with 0s.

Undo:

- There is no undo

Write this block to disk if we had to 0 it.

INODE_UPDATE_TYPE

Redo:

- If the current `sfi_type` is the journalled old type, set it to the new type. Otherwise do nothing.

Undo:

- If the current `sfi_type` is the journalled new type, set it to the new type. Otherwise do nothing.

Write this inode back to disk

Journalled Operations

`sfs_balloc` (actually journalled in `sfs_clearblock`): All block allocations are journalled here. It was more convenient to put journalling code in `sfs_clearblock` since we needed to know the buffer that was created for checkpointing and trimming purposes

`sfs_bfree` (actually journalled in `sfs_bfree_prelocked`): All block deallocations are journalled here.

`sfs_blockobj_set`: This is a metadata update to an inode or other object

`sfs_discard_subtree`: This required multiple journal entries to log disk addresses being cleared at each indirection level (metadata update). Freeing is handled in `sfs_bfree`.

`sfs_itrunc`: The resize is journalled. All other changes are handled by `sfs_discard_subtree`.

`sfs_load_vnode`: Type changes when forcetype is set are journalled.

`sfs_partialio`: Block writes are journalled.

`sfs_blockio`: Same as sfs_partialio

`sfs_io`: File size changes are journalled. All actual I/O is journalled by either `sfs_partialio` or `sfs_blockio`

`sfs_metaio`: Metadata changes and file size changes are journalled.

`*->sfi_linkcount changes`: There are about 20 locations where an inode's link count is modified with any of

`*->sfi_linkcount++`

`*->sfi_linkcount--`

`*->sfi_linkcount += 2`

`*->sfi_linkcount -= 2`

Transactions

In addition to log records, we introduce transactions for more efficient log record management. We use each process's pid for transaction id. Since user processes are single threaded, we don't need to worry about id collision. However, there will be multiple records with same transaction id in the journal. But they will also be handled correctly during recovery since each transaction contains a `TRANS_BEGIN` and `TRANS_COMMIT` record. Each log record will be associated with its corresponding transaction id. `TRANS_BEGIN` and `TRANS_COMMIT` record will be written at the beginning and the end of each higher-level sfs function (sfs_creat, sfs_remove etc).

All active transactions will be added to a global array called the transaction_active. A transaction will be removed from that list once it's committed. We designed an API for beginning/committing a transaction, `sfs_begin_trans` and `sfs_commit_trans`, where a transaction is added/removed from the `active_transactin` list after we write the corresponding long record to the journal.

To handle the major trap for lsn synchronization, the main logic of `sfs_begin_trans` is moved to `sfs_trans_callback`, where we actually create a transaction and add it to the active transaction list. The `sfs_begin_trans` simply calls `sfs_jphys_write` with `sfs_trans_callback`.

There are 9 types of transactions we track:

- `TRANS_WRITE`
- `TRANS_TRUNCATE`
- `TRANS_CREAT`
- `TRANS_LINK`
- `TRANS_MKDIR`
- `TRANS_RMDIR`
- `TRANS_REMOVE`
- `TRANS_RENAME`
- `TRANS_RECLAIM`

WAL Enforcement & Metadata

In order to ensure that all journal entries make it to disk before their associated buffers do, we track additional metadata in each buffer and added hooks in the functions responsible for buffer writes to flush the associated journal entries.

We introduced the `b_fsdata` struct, which contains the following fields:

`sfs`: The sfs filesystem this buffer belongs to

`diskblock`: The block number of the block that backs this buffer

`oldest_lsn`: *The lsn of the journal entry of the earliest operation to modify this buffer after this buffer was written to disk and marked clean. When checkpointing, we make sure not to trim past this value for any of the currently dirty buffers.

`newest_lsn`: The lsn of the journal entry of the most recent operation to modify this buffer. Whenever a buffer is written to disk, we force the journal to flush all journal entries up to this newest lsn.

`buf`: A reference back the the metadata belongs to. Used only for debugging.

* Not used for WAL enforcement, but included in this section for completeness. See Checkpointing

sfs_writeblock

Whenever a non-null `fsbufdata` is provided, we can assume that this is being called from `buffer_writeout_internal`. In this case, we flush the journal up to the `newest_lsn` referenced by the metadata and reset both `oldest_lsn` and `newest_lsn` to 0. This causes subsequent journalling actions to set these values when this buffer is modified again.

sfs_freemapio

We modified the `sfs_fs` struct to track an `oldest_freemap_lsn` and `newest_freemap_lsn`. These are analogous to the `oldest_lsn` and `newest_lsn` in the buffer metadata, but these track the earliest lsn of an operation that modified the freemap after it was written, and the lsn of the most recent operation that modified the freemap. When `sfs_freemapio`, we flush the journal to `newest_freemap_lsn` and reset both values to 0. During checkpointing, we make sure not to trim past `oldest_freemap_lsn`.

Checkpointing

We add checkpointing logic to ensure that the journal doesn't overflow. Checkpoint will be called when we're writing a record, and the odometer of the journal hits 1. Since we can't trim any record of an active transaction or dirty buffer, we record a oldest_lsn field for each transaction and dirty buffer. During a checkpoint, we will go through the active transaction list and the dirty buffer list to calculate the oldest_lsn for us to trim. As noted in the previous section, we also compare the oldest_lsn with olest_freemap_lsn t make sure we don't trim pass any freemap related records.

We will call sfs_checkpoint twice during recovery: when we're done with recovering the journal and when we're done with recovering graveyard. The journal should be empty after we finish recovery.

Graveyard (Unlink/Reclaim)

To handle logic for reclaiming unlinked files, we decide to create a separate directory for graveyard.

mksfs

In mksfs, we hand-craft a new directory called graveyard. The logic is similar to the root directory and the graveyard will always live at inode 2. Graveyard directory lives inside root.

sfs_remove

In sfs_remove, we do not decrement the linkcount anymore. Instead, we directly link the file to the graveyard directory after it's removed from its parent. Decrementing linkcount and destroy the file will be completely handled by reclaim. Since graveyard is a directory, we could run into filename collision. In order to prevent that from happening, we rename each file based on its ino number before moving it into the graveyard. Since no 2 active files could have same inode number, collision will not happen.

sfs_reclaim

In sfs_reclaim, we try to find the file in graveyard. If didn't we find it in the graveyard, it mean the file is closed but not removed. No file removal logic is necessary. If we find the file in graveyard, it means that reclaim is called during a sfs_remove. We then decrement the linkcount , remove the file from graveyard, and call sfs_iturnc if linkcount is 0.

Recovery

During recovery, we load up the graveyard and go through its entries and try to call sfs_relcaim on each of them. Since this will create additional journal records, we take another checkpoint after graveyard recovery is done.

Testing

We wrote a script to automate the testing of our journalling code. Before running any test, our script would

1. Fill the disk image with poisson data using `hostbin/host-poissondisk`
2. Format the disk image using `hostbin/host-mksfs`

We then ran:

1. Every test in `frack` using doom counters from 1 to 4
2. `dirconc` with doom counters from 1 to 19
3. `bigfile` with doom counters from 1 to 19
4. `rmtest`

After running `frack do *`, we would immediately run `frack check *` to verify that our recovery put the filesystem in a consistent state. In all cases we then ran `hostbin/host-sfsck` to verify that there were no filesystem errors.