CS161 Assignment 4

Abby Lyons, Timothy Tamm

Table of Contents

Introduction

This time, we actually decided to describe each section of the problem set at the start of each major header below. Hopefully that means less reading for you!

Journaling

Each high-level file system call has its own kind of transaction, each of which contains a start record, an end or abort record, and any number of other records in between (see below). They also each have their own transaction number, determined by the log sequence number of the start record. The transaction number is stored in the current thread, and open transactions are additionally stored in an array for easier checkpointing.

Overview of records

Here is a list of all of the different kinds of records:

- X: start transaction x
- Y: end transaction x
- Z: abort transaction x
- 1. change direntry d for inode x from oldName, oldIno to newName, newIno, for transaction tnx
- 2. zero out block b, for transaction tnx
- 3. mark block b in the freemap for metadata (allocate block b), for transaction tnx
- 4. unmark block b in the freemap (free block b), for transaction tnx
- 5. change sfi_size of inode x from y to z, for transaction tnx
- 6. Change sfi_linkcount for inode x from y to z, for transaction tnx
- 7. Change n-indirect pointer for inode x from ino y to ino z (n is 1, 2 or 3), for transaction tnx
- 8: Change the n-th ino in the indirect inode x from y to z , for transaction tnx
- 9. Change the n-th direct pointer for inode x from ino y to ino x, for transaction tnx
- 10. Write data with checksum c into block b, for transaction tnx
- 11. Change type of inode x from y to z, for transaction tnx

And below is a list of SFS functions to update:

- sfs_dir_link: 1
- sfs_dir_unlink: 1
- sfs_clearblock: 2
- sfs_balloc: 3
- sfs_bfree_prelocked: 4
- sfs_io: 5
- sfs_partialio: 10

- sfs_blockio: 10
- sfs_metaio: 5
- sfs_write: X,Y, Z
- sfs_truncate: X, Y, Z
- sfs_blockobj_set: 7
- sfs_discard_subtree: 8 (in every place where a layer's data is modified)
- sfs_discard: 9
- sfs_itrunc: 5
- sfs_creat: X, Y, Z, 6
- sfs_link: X, Y, Z, 6
- sfs_mkdir: X, Y, Z, 6
- sfs_rmdir: X, Y, Z, 6
- sfs_remove: X, Y, Z, 6
- sfs_rename: X, Y, Z, 6, 1
- sfs_loadvnode: 11

Code for record data structures

```
#define START_TRANSACTION 1
#define END_TRANSACTION 2
#define ABORT_TRANSACTION 3
#define CHANGE_DIRENTRY 4
#define ZERO_BLOCK 5
#define ALLOC_BLOCK 6
#define FREE_BLOCK 7
#define CHANGE_SIZE 8
#define CHANGE_LINK_CNT 9
#define CHANGE_INDIRECT_PTR 10
#define CHANGE_DIRECT_PTR 11
#define CHANGE_INO_IN_INDIRECT 12
#define WRITE_BLOCK 13
#define CHANGE_INODE_TYPE 14

/* high-level fs functions that need logging */
typedef enum {
    P_WRITE,
    P_RECLAIM,
    P_TRUNCATE,
    P_CREAT,
    P_MKDIR,
    P_LINK,
    P_RMDIR,
    P_RENAME,
    P_REMOVE,
    P_MORGUE
} fs_logfunc_t;

/* Start/End/Abort Transaction */
struct transaction_le {
    sfs_lsn_t le_tnx;       /* Transaction id*/
    fs_logfunc_t le_func;   /* Which function started this transaction */
};

/* Change Directory entry */
struct change_direntry_le {
    sfs_lsn_t le_tnx;               /* Transaction id */
    uint32_t le_ino;                /* Inode number */
    uint32_t le_direntry;           /* Directory entry to change */
    uint32_t le_oldino;             /* Old inode in the direntry */
    char le_oldname[SFS_NAMELEN];   /* Old name in the direnty */
    uint32_t le_newino;             /* New inode in the direntry */
    char le_newname[SFS_NAMELEN];   /* New name in the direnty */
};

/* free/zero/alloc block */
struct block_le {
    sfs_lsn_t le_tnx;       /* Transaction id */
    uint32_t le_blocknum;   /* number of the block to modify */
};

struct change_block_obj_le {
    sfs_lsn_t le_tnx;       /* Transaction id */
    uint32_t le_blocknum;   /* number of the block to modify */
    uint32_t le_offset;     /* offset into the block object */
    uint32_t le_oldval;     /* old value in the block object */
    uint32_t le_newval;     /* new value in the block object */
};

/* Change Size */
struct change_size_le {
    sfs_lsn_t le_tnx;       /* Transaction id */
    uint32_t le_ino;        /* Inode number */
    uint32_t le_oldsize;    /* Old size of the file */
    uint32_t le_newsize;    /* New size of the file */
    uint32_t le_type;       /* type of the inode */
};

/* Change Linkcount */
struct change_linkcount_le {
    sfs_lsn_t le_tnx;       /* Transaction id */
    uint32_t le_ino;        /* Inode number */
    uint16_t le_oldcount;   /* Old linkcount of the inode */
    uint16_t le_newcount;   /* New linkcunt of the inode */
    unsigned le_inodetype;  /* the type of the object whose linkcount we're modifying */
};

/* Change Indirect Ptr */

typedef enum {
    P_SINGLE,
    P_DOUBLE,
    P_TRIPLE
} indirection_level_t;

struct change_indirect_ptr_le {
    sfs_lsn_t le_tnx;               /* Transaction id */
    uint32_t le_ino;                /* Inode number */
    indirection_level_t le_level;   /* which indirect ptr to change */
    uint32_t le_oldptr;             /* ino of the old block it pointed to */
    uint32_t le_newptr;             /* ino of the new block it will point to */
    uint16_t le_type;               /* type of the inode */
};

/* Change Direct Ptr */
/* Change Ino in Indirect */
struct change_ptr_le {
    sfs_lsn_t le_tnx;           /* Transaction id */
    uint32_t le_ino;            /* Inode number */
    uint32_t le_ptrnum;         /* which direct ptr to change */
    uint32_t le_oldptr;         /* ino of the old block it pointed to */
    uint32_t le_newptr;         /* ino of the new block it will point to */
    uint16_t le_type;           /* type of the inode */
};

/* Write Block */
struct write_block_le {
    sfs_lsn_t le_tnx;       /* Transaction id */
    uint32_t le_block;      /* block that is being written into */
    uint32_t le_checksum;   /* Checksum of the written data */
};

/* Change Inode Type */
struct change_inode_type_le {
    sfs_lsn_t le_tnx;           /* Transaction id */
    uint32_t le_ino;            /* Inode number */
    uint16_t le_oldtype;        /* Old type of the inode */
    uint16_t le_newtype;        /* New type of the inode */
};

/* container-level record types (allowable range 0-127) */
#define SFS_JPHYS_INVALID       0           /* No record here */
#define SFS_JPHYS_PAD           1           /* Padding */
#define SFS_JPHYS_TRIM          2           /* Log trim record */
#define START_TRANSACTION       3
#define ABORT_TRANSACTION       4
#define END_TRANSACTION         5
#define CHANGE_DIRENTRY         6
#define ZERO_BLOCK              7
#define ALLOC_BLOCK             8
#define FREE_BLOCK              9
#define CHANGE_SIZE             10
#define CHANGE_LINK_CNT         11
#define CHANGE_INDIRECT_PTR     12
#define CHANGE_DIRECT_PTR       13
#define CHANGE_INO_IN_INDIRECT  14
#define WRITE_BLOCK             15
#define CHANGE_INODE_TYPE       16
#define CHANGE_BLOCK_OBJ        17
```

Outline for writing records

Write_record (in sfs_logging.c)
This does the bulk of the work for writing records to the journal. It requires the sfs, the record type, and a variable number of other arguments.

- Create one of each type of record struct. (one of the unfortunate parts of using a switch statement and not wanting to call kmalloc)
- Switch on the record_type. For each record type, this involves setting all of the fields of the corresponding record type, a pointer to that record, and the length to the size of the record. There are a couple of special cases:
    - START_TRANSACTION: acquire the **sfs_recordlock** (which will help us hold a particular LSN so we can record it as the current transaction number in the record), call **sfs_jphys_peeknextlsn**, and update the current process's transaction to that value.
    - CHANGE_DIRENTRY: this involves a couple of calls to **strcpy**
- Call **sfs_jphys_write**.
- If it was an END_TRANSACTION, set the current thread's transaction number to 0, and remove it from the active transaction list.
- Release the record lock if it was acquired.
- Do proper synchronization to wake up the checkpointer, if necessary. (See section on checkpointing.)
- Return the lsn returned from **sfs_jphys_write**.

For each of the records listed in the previous section, the following things might also need to happen outside of the **write_record** call:

- Calculate the checksum for data being written
- Update the buffer metadata to include the newest LSN (after acquiring the metadata lock)
- Update the freemap metadata to include the newest LSN (after acquiring the freemap lock)

Checkpointing

We will be implementing rolling checkpoints by finding the oldest LSN of incomplete transactions and dirty freemap/buffers, and trimming everything older than that LSN. The freemap and buffers will hold metadata that includes the oldest and newest LSN that modified them (oldest is for checkpointing, newest is for flushing the journal).

Synchronization will involve a lock and CV, as well as the journal odometer. The checkpointing thread will check the odometer, and if it hasn't reached the checkpoint bound, it locks and waits on the CV, and immediately unlocks after it's woken up. In **write_record**, if the checkpoint bound has been reached, it broadcasts on the CV.

Data structures

Sfs.h will be updated to contain the following stuff:

```
/* new metadata struct for buffers and freemap */
struct sfs_metadata {
    sfs_lsn_t md_oldlsn;     /* LSN of oldest record that wrote here (0 if clean) */
    sfs_lsn_t md_newlsn;     /* LSN of youngest record that wrote here (0 if clean) */
};

/* this goes inside the currently existing struct */
struct sfs_fs {
    struct sfs_metadata *sfs_freemap_metadata;  /* metadata for freemap */

    /* stuff for checkpointing */
    struct lock *sfs_checkpoint_lk;             /* lock for CV below */
    struct cv *sfs_checkpoint_cv;               /* wakes up checkpointing thread */
    sfs_lsn_t *sfs_checkpoint_bound;            /* checkpoint every n records */
    struct thread *sfs_checkpoint_thread;       /* checkpointing thread */
    struct proc *sfs_checkpoint_proc;           /* checkpointing proc */
    bool sfs_checkpoint_run;                    /* checkpointer will only run if true */
    struct lsnarray *sfs_active_tnx;            /* array of active transactions */
    struct lock *sfs_active_tnx_lk;             /* lock for array above */
    uint8_t sfs_in_recovery;                    /* true if in recovery */
};

/* function definitions */
void checkpoint_thread_init(struct sfs_fs *sfs);
void checkpoint_thread_f(void *data1, unsigned long data2);
void checkpoint(struct sfs_fs *sfs);
void update_buffer_metadata(struct buf *buffer, sfs_lsn_t tnx);
```

Existing functions to update

Sfs_attachbuf

- Create new buffer metadata, and set oldlsn and newlsn to 0.
- Attach it using **buffer_set_fsdata**.

Sfs_detachbuf

- Get the buffer metadata and free it.
- Remove the **KASSERT** at the end.

Sfs_writeblock

Note that buffer metadata will be marked dirty when records are written to the journal, if applicable. Here, we will want to mark the buffer metadata as clean.

- Before returning 0:
    - Acquire the metadata lock.
    - Update **fsbufdata** so oldlsn and newlsn are 0.
    - Release the metadata lock.

Sfs_sync_freemap

- Acquire the metadata lock.
- Update the freemap data so oldlsn and newlsn are 0.
- Release the metadata lock.

Sfs_mount and sfs_domount

- Initialize everything listed above, and start the checkpointing thread once recovery is over.

Sfs_unmount

- Uninitialize stuff if necessary, and set **sfs_checkpoint_run** to false so the checkpointing thread kills itself.

New functions

update_buffer_metadata

- Acquire the metadata lock.
- Get the given buffer's fsdata.
- Update the oldtnx if it's zero, and update the newtnx if it's less than the given tnx.
- Set the fs data.
- Release the metadata lock.

checkpoint_thread_f

This function will have a loop that checkpoints regularly. The thread that calls this function will be initialized at the end of sfs_domount, and it will be destroyed after sfs_unmount when **sfs_checkpoint_run** is set to false.

- While the checkpointer is supposed to run:
    - If the odometer hasn't reached the checkpoint bound yet, go to sleep.
    - Call checkpoint.
- Call **kern__exit**.

Checkpoint

This does just one round of checkpointing--I only abstracted it away like this so it can be called synchronously at the end of recovery.

- Call **sfs_jphys_peeknextlsn** to find the next LSN.
- Acquire the active transactions lock.
- Scan all active transactions, looking for the oldest LSN of incomplete transactions.
- Release the lock.
- Acquire the metadata lock.
- Scan all buffers, looking for the oldest LSN of dirty buffers. (This is actually a separate function in buf.c.) Update the current LSN to equal the return value of this if the return value is lower.

- Check the freemap metadata in **sfs->sfs_freemap_metadata**, and update the current LSN to equal the metadata's LSN if it's earlier than the currently existing one and nonzero.
- Release the metadata lock.
- Call **sfs_jphys_trim** to write the tail journal record at the current LSN.
- Clear the odometer.

Checkpoint_thread_init

- Fork **sfs_checkpoint_proc**.
- Fork the checkpointing thread so it calls **checkpoint_thread_f**.

Bufaray_find_oldest_dirty_lsn (in buf.c)

- Set a variable **lsn_keep** to the maximum value.
- Iterate through the dirty_buffers array.
  - Skip NULL buffers and those with different file systems.
  - Get the fs data. If it's not NULL and the **oldtnx** is less than **lsn_keep**, set **lsn_keep** to that value.
- Return **lsn_keep**.

Recovery

At a high level, we will have four phases for recovery: a check phase, which makes an array of protected blocks; an undo phase; a redo phase; a morgue phase, which decref's and invalidates everything in the morgue; and a checksum phase, which zeroes written blocks with different checksums.

We will build a main functions for handling journal records called: process_journal_record. This function takes in a journal record type, a void pointer containing the journal record data structure, a pointer to the sfs_fs we are operating on and an enum specifying the direction (P_UNDO, P_REDO). The function will switch on the record type and then process the journal record.

Data structures

We will assign a special inode slot on the disk to be the ondisk morgue. The morgue will be a directory inode and we will use the directory functions to move files into it after they've been unlinked in **sfs_remove** or **sfs_rmdir**. Every time an inode gets reclaimed in **sfs_reclaim**, we will check if it was in the morgue and if it was, we will remove it from the morgue.

Recovery code

The following will appear in sfs_fsops.c in lieu of the comment to “call your recovery code here”:

- Check phase
    - Start Reading the journal backwards, starting from the HEAD
    - Keep track of the latest allocation for each block to avoid overwriting user data during recovery
    - Keep track of all the end transactions. When you find a start transaction, check if we had found a matching end transaction. If not, then this transaction must have been aborted.
- Undo phase
    - Start Reading the journal backwards, starting from the HEAD
    - For every entry:
        - Get the type using sfs_jiter_type
        - Get the LSN using sfs_jiter_lsn
        - Get the data using sfs_jiter_rec
        - If it’s a write or zero block record, keep track of the block number and lsn in a protected block array if the blocknumber is not already in that array. This is necessary to avoid redoing zeroing on blocks that will later have userdata in them and to avoid checking the checksum on a block from an earlier write (we only want to do it on the last write).
        - Call process_journal_record passing in all the relevant data and the right direction
- Redo phase
    - Start Reading the journal forwards, starting from the TAIL
    - For every entry:
        - Get the type using sfs_jiter_type
        - Get the LSN using sfs_jiter_lsn
        - Get the data using sfs_jiter_rec
        - If it’s a write record, then skip, because checksumming will have already been done during the previous phase.
        - If it’s a zero_block record, then only do it if it was the last write or zero block that happened to the block (we keep track of this in the previous phase)
        - Call process_journal_record passing in all the relevant data and the right direction
- Morgue phase
    - See how big the morgue is using sfs_dir_nentries
    - For every entry:
        - Load the entry in using sfs_readdir

        - Create a sfs_vnode that points to the inode in the entry using sfs_loadvnode
        - Unlink the node from the morgue
        - Reclaim the node
    - Truncate the morgue
    - Issue a checkpoint

Finally, kick off the checkpointing thread.

Process_journal_entry

Process_journal_entry will be a big switch function that switches based on the record type and then calls a helper function that handles undoing/redoing a specific record type. Most of these functions simply involve checking if the block is protected (to avoid overwriting user data), making sure, it’s not an aborted transaction when redoing, then reading the block in into a buffer, making the relevant change, marking the block dirty and then releasing the buffer.

The special case is parse_change_direntry. In order to leverage the power of finding the right block for a given direntry number, we use sfs_writedir. This however poses a challenge, because during the undo phase, we might try to write into a direntry that would be in a direct or indirect block, that’s not actually allocated. In order to get around this, we first check if the directory inode is allocated, then we load the inode and check if the type is correct and finally, we modified sfs_metaio, so that if sfs_writedir would still try to allocate a block during recovery, then we return a special error code (-1), that we check for in parse_change_direntry. If we get that error code, then that means that the relevant direct/ indirect block(s) never made it to disk, so we don’t have to undo anything.

Plan of Action

4/15-4/21
Saturday: enjoying last day of freedom
Sunday: implementing journal record structs, undo/redo for all journal entries
Monday: implementing checksums for user blocks, morgue init
Tuesday: getting wrecked by CS134
Wednesday: logging, dumpsfs pretty printer, fixing A3 stuff
Thursday: logging, adding files to morgue, checking the morgue in sfschk
Friday: checkpointing

4/22-4/28
Saturday: checkpointing
Monday-Tuesday: debugging

Wednesday: testing
Thursday: more testing
Friday: even more testing
Saturday-Tuesday: more debugging and testing because we suck

(We will implement stuff and test stuff before the deadline. Who needs plans?)

Peer review notes

- Consider putting the checkpointing stuff in a different thread and having it wait on a semaphore
- We forgot loadvnode from our list of functions to update

Acknowledgements

We would like to thank Prof. James Mickens for teaching us the concepts, Alex Patel and team GeckOs for helping us refine our design, and all of the TFs for being super helpful and available and awesome :)