Jack Dent
Graham Lustiber

CS161, Assignment 4 Design Document

Contents

1) Introduction

The goal of this assignment is to implement a journaling system into the sfs file system so that os161 is able to recover from crashes and restore the integrity of the file system. This will also require a recovery algorithm to scan through the saved journal on disk and make repairs to the file system.

2) Transactions

Transactions are useful for checkpointing.

```
struct transaction {
     uint64_t t_start_lsn;
     uint64_t t_end_lsn;            // optional, may be useful
     uint32_t t_dirty_buf_count;    // 0 when transaction is done
     uint32_t t_volume;             // the volume the transaction
is relevant to
};
```

The following system calls from sfs_vnops.c modify the file system state and will thus require a transaction to be created:

```
sfs_reclaim()
sfs_creat()
sfs_mkdir()
sfs_write()
sfs_link()
sfs_rmdir()
sfs_remove()
sfs_rename()
sfs_truncate()
```

Active transactions will be stored in a global table of fixed sized size for the sake of checkpointing. Operations on this table will be protected by a transaction lock. They will also be stored in the calling thread's struct in a t_transaction field so that different sfs subroutines can access it.

3) Journaling

a. Record Schema

We support the following record types. To support undo/redo semantics, we must be able to track parameters for certain records which we accomplish by declaring structs. Parameters are indicated by the colon and are separated by commas.

- **Transaction records**
    - begin
    - commit

- **inode records**
    - get
    - update: new_inode, old_inode
    - release: inode_id

- **Directory records**
    - add
    - release: slot_id

- **Block records**
    - write: id, position, length

- **Freemap records**
    - alloc
    - release: id

A record contains an enum specifying the record type and a union of structs that may contain any necessary parameters. We can use C's internal union type as follows:

```
struct record {
      unsigned int transaction_id;
      enum r_type;
      union {
            struct inode_update r_inode_update;
            ...
      } r_parameters;
};
```

The records will be stored in a memory journal buffer and as well on disk via the jphys library. We will also store num_records on disk at the beginning of the disk journal so that we know on start-up whether any recovery has to be done.

We will need to handle freemap records slightly differently to all the other record types. Since the freemap is stored in memory and is not in the buffer cache, we will need to create a freemap record whenever we set or clear a bit in the freemap, since these writes will not be handled by the block write record.

N.B. we do not journal user data in the block records, which allows for some user data to be lost.

b. User Data Checksums

To avoid having to write out user data before logging it, we will perform a checksum hash on any user data being written out while it is in memory, storing the result in the record. At recovery time, we will do a checksum hash on the disk block. If the two hashes differ, we know the data did not make it out, so we must 0 out the disk block.

c. Checkpoints

We will run a background checkpoint thread for each disk loaded. When it is scheduled, the checkpoint routine will block all file system calls that would create new transactions. We will scan through all active transactions and find the lowest record number among them. We can then remove all record numbers below this minimum, thus freeing up space in our journal. When this is done, unblock all the waiting file system calls and yield. Also clean any dead files from the graveyard that have been successfully removed on disk.

Synchronization: we will use the transaction lock and hold it for the duration of a checkpoint to ensure no new transactions come in.

d. Graveyard

In the operations unlink(), rename(), and rmdir(), it is possible to have an intermediate state where a directory or file is not connected to the sfs directory tree, but is still taking up blocks on disk. Thus, we add it to an array of dead files/directories on disk, logging this with an add_to_graveyard record. Thus, on boot up, we know to truncate these files to free the blocks. Checkpointing will also clean this graveyard, periodically. The graveyard will be initialized in makesfs() in the first block after the end of the journal region

4) Recovery

Steps:

1) REDO/UNDO phase:
    a) We first make one scan through the journal records, noting which transactions have their commits. We also keep track of which blocks had user data written to them, and whether they were “successful” for not
    b) We make a REDO pass through the journal records, applying each operation to the disk. If we don’t have a “successful” record for a user block write, we must 0 it out. Moreover, if the last write to a block was userdata, we ignore the previous writes to it in order to avoid overwriting user data with metadata
    c) We make an UNDO pass, undoing any records that belong to transactions for which we don’t have a commit record.
2) Perform a checkpoint in order to trim the journal to 0
3) Handle the graveyard by loading each vnode and then closing it, this will truncate each file to 0
4) Perform another checkpoint

Synchronization is not needed since a single kernel thread will be the only thing with access to the disk at this point

a. Crashing while recovering

Because all of our records will be idempotent operations, crashing again while running the recovery operation should have no effect on integrity of the file system. In the worst case, certain operation may be applied multiple times.

5) Integrating with SFS

The system calls listed above under Transactions will be responsible for allocating a transaction struct, and saving it to curthread->t_transaction, whereby it can be access by subroutines in sfs. The last sfs function will send a commit record and then set curthread->t_transaction back to NULL.

6) Bootstrapping

Whenever a disk is mounted, we will call a recovery() routine that performs the above file system restoration. Once we know we are recovered, we will launch the checkpointing thread to run in the background.

7) Plan of Action

Sun 4/17-Thur 4/21 - Work on journaling
Fri 4/22-Sat 4/24 - Implement recovery code
Mon 4/25-Fri 4/29 - Testing