Computer Science 161: Operating Systems
Assignment 4 Design Document[1]

David Li and Garrett Tanzer
April 13, 2017

1. Introduction

Assignment 4 can be divided relatively cleanly into two major, interrelated components: logging and recovery atop SFS's `jphys` journal.

We will implement write-ahead redo-undo logging, with rolling checkpoints at a frequency to be determined by tuning. Specifically, our scheme will use low- to mid-level physical record journaling. That is to say, multiple log entries may be coupled into a single, idempotent transaction representing some higher-level logical function. As designated by `jphys`, the entries will be stored in a circular buffer, ordered by monotonically increasing log sequence numbers (LSNs). Our specific journal entry types are described in 2.1.

The recovery process will automatically begin when booting from a disk that has not been properly unmounted. It will entail four passes: the first finds logs acting on what used to be metadata but is physically now user data, the next two perform traditional redo/undo loops, then the final pass ensures that stale (uninitialized) data isn't surfaced to the user level for security reasons.

There are several ancillary functions that will also be discussed below, like tracking for unlinked but not yet reclaimed files and in-memory transaction management.

2. Overview

2.1 Journaling

General journaling policy will be implemented as described in the Introduction—generally speaking, write-ahead redo-undo logging. The write-ahead property will be guaranteed by calling `sfs_jphys_flush` on the logs for a transaction before issuing the in-place write. The LSN it flushes up to (inclusive) will be decided using per-volume metadata attached to each buffer (the

[1] Thanks to DodOS for ideas/revisions for record types, helper function records, reserved directory inode for unlinked files, checkpointing twice in the recovery process, the method for ensuring write-ahead for the block freemap, and various other small changes.

most recent LSN associated with a change to the buffer). Because the block freemap is not a buffer, we will have to keep track of similar information ourselves and enforce write-ahead logging for it separately.

Basically, in-place writes are only allowed after the journal entry has been written to disk. After all the in-place writes are issued, the TxEnd is written; so, when TxEnd reaches the physical journal, there is no guarantee about the state of the in-place write. Checkpointing will be described in 3.3 and will deal with this uncertainty.

Specific additional policies include:

- There will be no explicit transaction start. The start will simply be the first operation with a given LSN.
- There will be no explicit transaction abort tag. Aborted transactions will log their error path and commit normally with TxEnd.
- We do not handle the case where a single transaction is big enough to wrap around the entire journal.
- Interleaving of transactions in the journal is allowed; no part of recovery will be dependent on contiguity.
- Generally, logs will be issued before calls to `buffer_mark_dirty` (and modifications to the block freemap)
- Only `jphys`'s writer mode will ever be activated once the file system has been mounted.

Here are our anticipated record types and their struct definitions. Some have implicit rather than explicit redo or undo commands because one value is necessarily zero—but they all must be performable in both directions, and idempotent if repeatedly redone, undone, or both. We will probably have a helper function to issue each type of record.

```
0-2  ...
3    SFS_JPHYS_TXEND
4    SFS_JPHYS_ALLOCB
5    SFS_JPHYS_FREEB
6    SFS_JPHYS_WRITEB
7    SFS_JPHYS_WRITE16
8    SFS_JPHYS_WRITE32
9    SFS_JPHYS_WRITEM
10   SFS_JPHYS_WRITEDIR
```

Transaction End: marks end of transaction, includes transaction type (for debugging)

```
struct sfs_jphys_txend {
      uint8_t type;
};
```

Allocate Block: allocate a block in the freemap at `index`

```
struct sfs_jphys_allocb {
      daddr_t index;
};
```

Free Block: free a black in the freemap at `index`

```
struct sfs_jphys_freeb {
      daddr_t index;
};
```

Write to (User) Block: write user data to a block with a checksum to find stale blocks at recovery

```
struct sfs_jphys_writeb {
      uint64_t checksum;
      daddr_t index;
};
```

Write 16 Bits: write 16 contiguous metadata bits, probably for inode linkcount or type changes

```
struct sfs_jphys_write16 {
      daddr_t index;
      uint16_t old;
      uint16_t new;
      uint16_t offset;
};
```

Write 32 Bits: write 32 contiguous metadata bits, probably for inode size changes

```
struct sfs_jphys_write32 {
      daddr_t index;
      uint32_t old;
      uint32_t new;
      uint16_t offset;
};
```

Mass Write (to Metadata): write up to 249 metadata bytes just in case we need coarse changes for some reason

```
struct sfs_jphys_writem {
      daddr_t index;
      uint16_t offset;
      char old[249];
      char new[249];
```

```
};
```

Write Directory Entry: write an entry into a directory (will be used to either add or remove items from directories by making `old` or `new` represent empty direntries)

```
struct sfs_jphys_writedir {
	daddr_t index;
	uint32_t slot;
	struct sfs_direntry old;
	struct sfs_direntry new;
};
```

2.2 Recovery

As mentioned in the introduction, recovery will take place over four passes, each following the iteration model provided for `jphys`: This will occur in `jphys`'s reader mode.

```
sfs_jiter *ji;

result = sfs_jiter_fwdcreate(sfs, &ji);
if (result) fail;
while (!sfs_jiter_done(ji)) {
	type = sfs_jiter_type(ji);
	lsn = sfs_jiter_lsn(ji);
	recptr = sfs_jiter_rec(ji, &reclen);
	...
	result = sfs_jiter_next(sfs, ji);
	if (result) fail;
}
sfs_jiter_destroy(ji);
```

The first pass will go forwards over the journal, noting in a bitmap which blocks operated on represent user data at the end of the journal.

The second pass will go forwards over the journal, redoing all committed transactions, unless they modify pages that appear in the bitmap created by the first pass.

The third pass will go backwards over the journal, undoing all uncommitted transactions, unless they modify pages that appear in the bitmap created by the first pass.

The fourth pass will go backwards over the journal and, for the last completed creation/append of a new user sector, check the record's checksum against the checksum generated by the sector of storage. If the checksums match, zero the storage.

Checkpoint at this step, since now all journal records are reflected in consistent metadata.

Finally, reclaim all unlinked files stored in the structure described in 3.1.

Checkpoint again to clear the records from reclaiming all the unlinked files.

After the recovery process runs, all metadata should be consistent, and there should be no storage leaks in the block freemap. The entire recovery process should be idempotent, such that if the system crashes while it is occurring, the disk state will be recoverable using the original journal.

3. Topics

3.1 Unlink/Reclaim

When a file is unlinked, remove it from its directory as expected and add it to a purgatory space on disk (a reserved directory inode). When the file is reclaimed, remove it from that purgatory directory. Both of these operations should be logged in the journal.

This functionality will need to be added to the existing code for `sfs_dir_unlink` and `sfs_reclaim`. The directory inode will be initialized by a modified `mksfs`, and `dumpsfs` and `sfsck` will need to be updated to account for it. As mentioned above, at the end of recovery we will need to reclaim all files in purgatory.

3.2 Transactions

Transactions that haven't yet been checkpointed will be stored in an array.h list of `transaction` structs, where the struct is defined as the following:

```
struct transaction {
      uint64_t lsn;
      uint8_t nbufs;
      bool txend;
};
```

Because LSNs are monotonically increasing, this list will always be sorted, so we could use binary search or something similar to find the transaction associated with a particular LSN if we so choose. This list (and the transactions in it) will be protected by a sleeplock. Each process's current LSN will be stored in its `proc` struct so that it can be accessed with `curproc`.

Each `transaction` will be created in a callback function from `sfs_jphys_write` to avoid race conditions ("major trap" in 12 of April 5 section notes). `txend` is initialized to `false` and is set to be `true` once the transaction end is flushed to the journal on disk.

When buffers are dirtied, we will add file specific metadata (`b_fsdata`) pointing to the relevant transaction and increase `nbufs`. When the buffer is flushed out to disk, `nbufs` will be decremented. (Both these operations must be protected by the transaction array's sleeplock.) When `nbufs` reaches 0 (and `txend` is `true`), the transaction's state is reflected on disk, and it can be checkpointed out of the journal.

3.3 Checkpoints

Checkpoints will be triggered by the in-memory transaction bookkeeping system when it detects the first *x* transactions have been fully written to disk, with *x* to be tuned at implementation. This constitutes a relatively natural implementation of rolling checkpoints, assuming similar behavior to the buffer cache. If information isn't reaching disk quickly enough (and the journal is filling up more quickly than it is emptied), we may add forcible flushing triggered by the creation of new transactions.

Checkpointing will be implemented using `sfs_jphys_trim`. If the checkpoint will trim out all transactions in the journal, we'll also need to use `sfs_jphys_peeknextlsn`. Peek at the next LSN before scanning through the transaction list to avoid race conditions ("minor trap" in 10.4 of April 5 section notes).

3.4 Transaction-Issuing Operations

The following functions will represent different types of transactions (so in implementation, they will need to create a TxEnd before returning):

- `sfs_dir_unlink();`
- `sfs_reclaim();`
- `sfs_write();`
- `sfs_truncate();`
- `sfs_creat();`
- `sfs_mkdir();`
- `sfs_link();`
- `sfs_rmdir();`
- `sfs_rename();`

3.5 Record-Issuing Operations

The following places will issue records within each transaction:

- `sfs_balloc()`:
  - Issues `SFS_JPHYS_ALLOCB`
- `sfs_bfree_prelocked()`:
  - Issues `SFS_JPHYS_FREEB`
- `sfs_blockobj_set()`:
  - Issues `SFS_JPHYS_WRITEM` with new and old block data
- `sfs_discard_subtree()`:
  - Issues `SFS_JPHYS_WRITE32` to appropriately handle changes to the pointer tree structure
- `sfs_partialio()`:
  - Issues `SFS_JPHYS_WRITEB`, calculating the checksum
- `sfs_blockio()`:
  - Issues `SFS_JPHYS_WRITEB`, calculating the checksum
- `sfs_writedir()`:
  - Issues `SFS_JPHYS_WRITEDIR` with new and old directory entries
- `sfi_linkcount` updates:
  - Any time `sfi_linkcount` is updated somewhere not mentioned above, we will issue an `SFS_JPHYS_WRITE16` with appropriate offset into the inode
- `sfi_type` updates:
  - Any time `sfi_type` is updated somewhere not mentioned above, we will issue an `SFS_JPHYS_WRITE16` with appropriate offset into the inode
- `sfi_size` updates:
  - Any time `sfi_size` is updated somewhere not mentioned above, we will issue an `SFS_JPHYS_WRITE32` with appropriate offset into the inode

3.6 Synchronization

The big synchronization problems have already been discussed in other sections (callback function pointer, etc.). Since the handout code is mostly atomic already, we mostly need to make sure that we don't nest functions in ways such that they try to reacquire locks or enter subfunctions with the wrong locks held.

4. Plan of Action

Monday:	struct/macro implementations, finish syscall merge
Tuesday:	transactions, transaction-issuing operations
Wednesday:	catch-up
Thursday:	record-issuing operations
Friday:	catch-up
Saturday:	checkpointing
Sunday:	unlink/reclaim
Monday:	recovery operations
Tuesday:	recovery passes
Wednesday:	catch-up
Thursday:	test
Friday:	test

Code-Reading Questions

1 ---

When sys_remove is called on a file that’s open in another process, it is only unlinked, not reclaimed, so that other process will have normal read/write access to that file until it is closed (and the file will exist somewhere until all file descriptors to it are closed).

2 ---

VOP_ instructions are macros for sfs_ functions, so that’s as low as we need to go for this one. VOP_DECREF() actually isn’t an sfs function but does VOP_RECLAIM (sfs_reclaim()) when refcount reaches 0.

Indentation matters, so hopefully the formatting works.

a ---

```
sys_open()
        vfs_open()
                {
                        vfs_lookparent()
                                VOP_LOOKPARENT()
                                VOP_DECREF()
                        VOP_CREAT()
                        VOP_DECREF()
                } or {
                        vfs_lookup()
                                VOP_LOOKUP()
                                VOP_DECREF()
                }
                VOP_EACHOPEN()
                {
                        VOP_DECREF()
                } or {
                        VOP_TRUNCATE()
                        VOP_DECREF()
                }
```

The meat of this function is either VOP_CREAT() or VOP_LOOKUP(), depending on input

b --

```
sys_write()
    VOP_WRITE()
    VOP_ISSEEKABLE()
```

c --

```
sys_mkdir()
    vfs_mkdir()
        vfs_lookparent()
            VOP_LOOKPARENT()
            VOP_DECREF()
        VOP_MKDIR()
        VOP_DECREF()
```

3 --

Error handling calls are omitted

a --

The relevant sfs call here is sfs_creat().
sfs_creat() opens a file that already exists, or creates a new one at the specified path and opens it.

```
sfs_creat()
    reserve_buffers()
    sfs_dinode_load()
        buffer_read()
    sfs_dinode_map()
        buffer_map()
    sfs_dinode_unload()
        buffer_release()
    sfs_dir_findname()
        sfs_dir_nentries()
            sfs_dinode_load()
                buffer_read()
```

```
        sfs_dinode_map()
            buffer_map()
        sfs_dinode_unload()
            buffer_release()
    sfs_readdir()
        sfs_metaio()
            sfs_dinode_load()
                buffer_read()
            sfs_dinode_map()
                buffer_map()
            sfs_bmap()
                sfs_get_indirection()
                sfs_dinode_load()
                sfs_blockobj_init_inode()
            buffer_read()
            buffer_map()
            sometimes buffer_mark_dirty()
            sometimes sfs_dinode_mark_dirty()
                buffer_mark_dirty()
{
    sfs_loadvnode()
        buffer_read()
        buffer_map()
        sometimes buffer_mark_dirty()
        sfs_vnode_create()
        buffer_release()
    unreserve_buffers()
} or {
    sfs_makeobj()
        sfs_balloc()
            sfs_clearblock()
                buffer_get()
                buffer_map()
                buffer_mark_valid()
                buffer_mark_dirty()
                sometimes buffer_release()
        sfs_loadvnode()
            see above
        sfs_dinode_load()
```

```
                buffer_read()
            sfs_dinode_map()
                buffer_map()
        sfs_dinode_map()
            buffer_map()
        sfs_dir_link()
            sfs_dir_findname()
                see above
            sfs_writedir
                sfs_metaio()
                    see above
        sfs_dinode_mark_dirty()
            buffer_mark_dirty()
        sfs_dinode_unload()
            buffer_release()
        unreserve_buffers()
    }
```

b --

```
sfs_write()
    reserve_buffers()
    sfs_io()
        sfs_dinode_load()
            buffer_read()
        sfs_dinode_map()
            buffer_map()
        sometimes sfs_partialio()
            sfs_bmap()
                sfs_get_indirection()
                sfs_dinode_load()
                    buffer_read()
                sfs_blockobj_init_inode()
                sfs_bmap_subtree()
                    sfs_bmap_get()
                    buffer_read()
                    etc.; it's a loop
                sfs_blockobj_cleanup()
                sfs_dinode_unload()
```

```
                    buffer_release()
            buffer_read()
            buffer_map()
            buffer_mark_dirty()
            buffer_release()
        sfs_blockio()
            sfs_bmap()
                see above
            buffer_get()
            buffer_map()
            buffer_mark_valid()
            buffer_mark_dirty()
            buffer_release()
        sometimes sfs_partialio()
            see above
        sometimes sfs_dinode_mark_dirty()
            buffer_mark_dirty()
    unreserve_buffers()
```

c --

```
sfs_mkdir()
    reserve_buffers()
    sfs_dinode_load()
        buffer_read()
    sfs_dinode_map()
        buffer_map()
    sfs_dir_findname()
        see above
    sfs_makeobj()
        see above
    3x sfs_dir_link()
        see above
    2x sfs_dinode_mark_dirty()
        buffer_mark_dirty()
    2x sfs_dinode_unload()
        buffer_release()
    unreserve_buffers()
```

4 --

sfs_mkdir() calls some buffer functions repeatedly, so I assume this question is just asking us to trace through reserve_buffers(), buffer_read(), buffer_map(), buffer_mark_dirty(), buffer_release(), and unreserve_buffers().

reserve_buffers()
    registers intent to use buffers for a file system operation

buffer_read()
    wraps buffer_read_internal() with buffer_lock
        buffer_get_internal()
            finds existing buffer, or makes a new one and attaches it
        buffer_readin()
            reads contents from disk into buffer with FSOP_READBLOCK
        buffer_release_internal()
            marks buffer no longer busy

buffer_map()
    returns pointer to buffer data

buffer_mark_dirty()
    marks the buffer as dirty

buffer_release()
    wraps buffer_release_internal() with buffer_lock
        marks buffer no longer busy, detaches metadata
        buffer is put on the end of the LRU list, from where it will eventually be written to disk with FSOP_WRITEBLOCK

unreserve_buffers()
    releases buffer reservation

5 --

a. safe, done in sfs_reclaim()
b. safe, done in sfs_reclaim()
c. unsafe because getting a buffer can trigger a buffer eviction, which can trigger the journal
d. safe, we see in sfs_mkdir() that locks are acquired parent > child

e. unsafe for same reason as d)

6 --

sfsck can fix directories with illegal sizes, nameless file entries, fileless name entries, etc.
sfsck can't fix directories without '.' or '..' entries
sfsck also can't fix a lot of problems with journal placement/bounds
sfsck also can't fix filesystems that aren't SFS (don't have its magic number)—who'd've guessed?

7 --

- If the system tries to append to a file, and the block allocation succeeds but the system crashes before the initial write goes through to disk, the user would have access to whatever data was last stored in that physical sector—however, our pass for stale data in recovery with checksums eliminates this problem

- If there's a transaction that commits, but some of the in-place writes to metadata hadn't gone through yet at the time of the crash, you'll have a state that reflects some but not all changes and is thus inconsistent (e.g. the block freemap reflects that a file has been truncated, but the file's size does not). The redo pass addresses this problem by making sure that all journaled records have their operations reflected on disk.

- If there's a transaction that hasn't been committed, but some of its records are written to the journal (i.e. the system crashes while it's ensuring write-ahead), the undo pass makes sure these changes aren't reflected on disk (because we don't have the complete transaction and transactions are atomic, we must make it as if the transaction never began)

- If the system crashes and there are logged operations on pages that were metadata at the time but have since been made user data (e.g. we modify a file's inode, unlink/reclaim that file, then an append to a file is placed in that physical page where the inode used to be), we have to make sure that our redo operations won't corrupt the user data (we don't have stored undo records for user data because we do record journaling, not block journaling). We solve this with our first pass in recovery by checking for sectors where this is the case, then later confirm each sector is still metadata before redoing operations on it