Computer Science 161: Operating Systems
Assignment 4 Design Document Draft

David Li and Garrett Tanzer
April 11, 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 to find logs acting on what used to be metadata but is physically now user data, then the two traditional redo/undo loops, then a final pass to ensure that stale (uninitialized) data isn't surfacing to the user level.

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. In-place writes are only allowed after the journal entry has been written to disk. After 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.

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.

```
SFS_JPHYS_ADDINODE       3    // add inode
SFS_JPHYS_REMINODE       4    // remove inode
SFS_JPHYS_APPEND         5    // file append
SJS_JPHYS_MAPPEND        6    // mass file append
SFS_JPHYS_TRUNC          7    // file truncate
SFS_JPHYS_MTRUNC         8    // mass file truncate
SFS_JPHYS_CH16           9    // change 16 bits
SFS_JPHYS_CH32           10   // change 32 bits
SFS_JPHYS_CHNAME         11   // change name
SFS_JPHYS_CHFREEMAP      12   // change free map slot
SFS_JPHYS_MCHFREEMAP     13   // mass change free map
SFS_JPHYS_DIRADDFILE     14   // add file to directory
SFS_JPHYS_DIRREMFILE     15   // remove file from directory
SFS_JPHYS_TXEND          127  // transaction end
```

`SFS_JPHYS_ADDINODE / SFS_JPHYS_REMINODE`

```
struct sfs_jphys_inode {
      uint32_t sfd_ino;
};
```

`SFS_JPHYS_APPEND / SJS_JPHYS_MAPPEND`

```
struct sfs_jphys_newpage {
      uint32_t index;
```

```
	uint32_t offset;
	uint32_t new;
	uint32_t checksum;
};
```

```
struct sfs_jphys_append {
	struct sfs_jphys_newpage n;
};
```

```
struct sfs_jphys_mappend {
	struct sfs_jphys_newpage ns[31];	// maximum in 504 bytes
};
```

SFS_JPHYS_TRUNC / SFS_JPHYS_MTRUNC

```
struct sfs_jphys_oldpage {
	uint32_t index;
	uint32_t offset;
	uint32_t old;
};
```

```
struct sfs_jphys_trunc {
	struct sfs_jphys_oldpage o;
};
```

```
struct sfs_jphys_mtrunc {
	struct sfs_jphys_oldpage os[42];	// variable length arrays?
};
```

SFS_JPHYS_CH16

```
struct sfs_jphys_ch16 {
	uint32_t index;
	uint32_t offset;
	uint16_t old;
	uint16_t new;
};
```

SFS_JPHYS_CH32

```
struct sfs_jphys_ch32 {
	uint32_t index;
	uint32_t offset;
	uint32_t old;
```

```
	uint32_t new;
};
```

SFS_JPHYS_CHNAME

```
struct sfs_jphys_chname {
	uint32_t sfd_ino;
	char old_name[SFS_NAMELEN];
	char new_name[SFS_NAMELEN];
};
```

SFS_JPHYS_CHFREEMAP

```
struct sfs_jphys_chfreemap {
	uint32_t index;
	uint8_t old;
	uint8_t new;
};
```

```
struct sfs_jphys_mchfreemap {
	struct sfs_jphys_chfreemap entries[63]; // 8 bytes each from padding
};
```

SFS_JPHYS_DIRADDFILE / SFS_JPHYS_DIRREMFILE

```
struct sfs_jphys_chfreemap {
	uint32_t dirinode;
	Uint32_t index;
	uint32_t fileinode;
	char name[SFS_NAMELEN];
};
```

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 temporary structure which operations act on metadata sectors that are user sectors at the end of the journal.

The second pass will go forwards over the journal, redoing all committed transactions, unless they appear in the temporary structure created by the first pass.

The third pass will go backwards over the journal, undoing all uncommitted transactions, unless they appear in the temporary structure 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.

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

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 (exact structure to be determined). When the file is reclaimed, remove it from that purgatory space. Both of these operations should be logged in the journal. 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 Journaled SFS Operations

`int sfs_dir_unlink(struct sfs_vnode *sv, int slot);`

To be described later

`int sfs_reclaim(struct vnode *v);`

To be described later

```
static int sfs_write(struct vnode *v, struct uio *uio);
```

To be described later

```
static int sfs_truncate(struct vnode *v, off_t len);
```

To be described later

```
static int sfs_creat(struct vnode *v, const char *name, bool
excl, mode_t mode, struct vnode **ret);
```

To be described later

```
static int sfs_mkdir(struct vnode *v, const char *name, mode_t
mode);
```

To be described later

```
static int sfs_link(struct vnode *dir, const char *name, struct
vnode *file);
```

To be described later

```
static int sfs_rmdir(struct vnode *v, const char *name);
```

To be described later

```
static int sfs_rename(struct vnode *absdir1, const char *name1,
struct vnode *absdir2, const char *name2);
```

To be described later

3.5 Recovery Operations

```
SFS_JPHYS_ADDINODE
```

To be described later

`SFS_JPHYS_REMINODE`

To be described later

`SFS_JPHYS_APPEND`

To be described later

`SJS_JPHYS_MAPPEND`

To be described later

`SFS_JPHYS_TRUNC`

To be described later

`SFS_JPHYS_MTRUNC`

To be described later

`SFS_JPHYS_CH16`

To be described later

`SFS_JPHYS_CH32`

To be described later

`SFS_JPHYS_CHNAME`

To be described later

`SFS_JPHYS_CHFREEMAP`

To be described later

`SFS_JPHYS_MCHFREEMAP`

To be described later

`SFS_JPHYS_DIRADDFILE`

To be described later

`SFS_JPHYS_DIRREMFILE`

To be described later

`SFS_JPHYS_TXEND`

To be described later

4. Plan of Action

Monday:
Tuesday:
Wednesday:
Thursday:
Friday:
Saturday:
Sunday:
Monday:
Tuesday:
Wednesday:
Thursday:
Friday:

Code-Reading Answers

- What happens (in broad terms) if sys_remove is called on a file that is currently open by another running process? Will a read on the file by the second process succeed? A write? Why or why not?
  - sys_remove calls vfs_remove, which calls vop_remove. This requires access to the vnode's refcount, which is protected by the spinlock, vn_countlock. Hence, a deletion is considered atomic and subsequent removals or writes will error.

- Describe the control flow, starting in the system call layer and proceeding through the VFS layer to reach SFS, that occurs for each of the following system calls. You need only trace the names of the functions that are called. Feel free to skip secondary or minor code paths that don't lead into SFS. (1 point)
    - SYS_open calls vfs_open on an uninitialized vnode pointer. vfs_open would call vfs_lookup (path, &vn) and then VOP_EACHOPEN on the found/created vnode. This then calls sfs_eachopen()
-