LFS on FUSE

Junxiao Shi, Karan Chadha

1 Introduction

Log-structured File System (LFS) [1] is a disk-based file system with high writing throughput. LFS is built with the assumption that most reads are satisfied by in-memory cache, so writes are the majority of disk activities. LFS writes to the disk sequentially, so that most disk bandwidth is used for writing data, and less time is spent on seeking.

We implement LFS as a userspace program. A log abstraction is created over a network-disk interface; files and directories are written to and read from the log. LFS mounts the filesystem in linux via FUSE [2], and supports UNIX semantics.

2 Physical Storage Format

This section defines the on-disk physical storage format of LFS.

2.1 Common Definitions

An **octet** is a unit of digital information that consists of eight bits.

Numbers are encoded in big endian, unless otherwise noted.

Timestamps are represented by the number of milliseconds since UNIX epoch.

Segment number is 32-bit zero-based. **Block number** is 32-bit zero-based, relative within a segment. **Segment-block number** is 64-bit, constructed by `(segment_number << 32) | block_number`.

Protocol Buffers [3] are a method of serializing structured data developed by Google. The length of a protocol buffer must be known when parsing it to a structure, therefore the length of a protocol buffer is stored before it as 32-bit big endian number.

2.2 Disk Layout

A *disk* is a block storage device. It is usually a mechanical device, or a *partition* when a partition table is created on the parent disk. A disk contains N_{sector} *sectors*, each of which can store LEN_{sector} *octets*; N_{sector} and LEN_{sector} are determined when the disk is created. Traditionally, the first sector of a disk is the boot sector reserved for machine code to boot the machine.

LFS divides a disk into N_{seg} *segments*, and divides each segment into N_{blk} *blocks*, each of which contains $N_{sector/blk}$ sectors; N_{seg}, N_{blk}, and $N_{sector/blk}$ are set when the filesystem is created on the disk, and satisfy:

$$N_{seg} \times N_{blk} \times SIZE_{blk} \leq N_{sector} \tag{1}$$
$$16 \leq N_{seg} \leq 2^{20} \tag{2}$$
$$8 \leq N_{blk} \leq 2^{12} \tag{3}$$
$$1024 \leq N_{sector/blk} \tag{4}$$
$$LEN_{blk} = LEN_{sector} \times N_{sector/blk} \geq 512 \tag{5}$$

Constraint 1 is obvious that the total number of sectors used by the filesystem cannot exceed the number of sector present on the disk. Other constraints are necessary for the correct operation and optimal performance of LFS.

In LFS, block is the smallest logical unit to store structures, and segment is the smallest unit of reading from or writing to the disk. A few segments at the beginning of the disk are called *system segments* and are used to store system-wide structures (Section 2.3; other segments are called *log segments* and are used to store the *log* (Section 2.4).

2.3 System Segment

A few segments at the beginning of the disk are used as system segments. Segment 0 contains the boot sector, and the *superblock* that contains static metrics of the filesystem. Two or more segments after segment 0 are used for two checkpoints.

2.3.1 Superblock

Superblock is a segment that contains static metrics of the filesystem. It is stored on segment 0 of the disk.

Block 0 contains the boot sector, and is never touched by LFS.

Block 1 contains the following 256-octet magic number:

```
f67ff220 7a499a8a 0a6ad1e3 27c6b21d 42e1e790 b13205fc 257589bc 1f7ab305
4f6c8dd5 e7c23be9 db76150f 94a7fac2 7d01ddfc f273715d ad7f88fd 5c209232
93806576 2118d1ab f6541cf5 548de4b6 cf1b3544 a12d908f 6ec074c6 a3e051f3
c103a44e 0ca99d3d 8d7d8f3f ea4d6a53 991a94da 982a9103 8a3e7d3f f1bc7513
6643d834 1934b64a 1714543c 9a186cf9 64d5eb6a fe2c72d5 e0113959 7b0395a2
e6a03dee 6f032615 d2dbe1ee cc54f775 b4e194b0 92415d13 f6c13267 729de3c3
53976f79 344fb120 4f40c2ba 189fec1c 9253f6a6 849d5bc9 4bbc08e5 7913dd46
618898fd 63f6366a 63dd7025 63a3d0a4 364ae264 8d82a6bb 0a0cbc22 84f4854f
```

When mounting the filesystem, LFS does not know $N_{sector/blk}$. In order to find the start of block 1, LFS MUST scan sequentially from the beginning of the disk, and find a sector that has this magic number as its first 256 octets. If the magic number is found at block SEC_{magic}, LFS knows $N_{sector/blk} = SEC_{magic}$. If the magic number is not found after scanning the first 1025 sectors, LFS concludes that the disk does not have a valid LFS structure, MUST NOT continue mounting, and SHOULD report an error.

The actual metrics are stored in block 2. It is encoded as a protocol buffer of the following structure:

```
message Superblock {
  optional fixed32 version = 1;//LFS version, contains '1'
  optional fixed32 noctet_sector = 2;//count of octets per sector
  optional fixed32 nsector_blk = 3;//count of sectors per block
```

```
  optional fixed32 nblk_seg = 4;//count of blocks per segment
  optional fixed32 nseg_disk = 5;//count of segments on the disk
  optional fixed32 first_logseg = 6;//first log segment number
  repeated fixed32 checkpoint_seg = 7;//start segment number of checkpoints
  optional fixed32 nseg_checkpoint = 9;//count of segments per checkpoint
  optional fixed32 nblk_segsummary = 8;//count of segment summary blocks per segment
}
```

All fields are required in this version; they are marked as "optional" so that a future specification can define a different format. `noctet_sector` = LEN_{sector}, `nsector_blk` = $N_{sector/blk}$, `nblk_seg` = N_{blk}, `nseg_disk` = N_{seg}. `checkpoint_seg` (exactly twice) and `first_logseg` must provided enough space for two checkpoint regions (Section 2.3.2). `nblk_segsummary` = $N_{segsummaryblk}$ (Section 2.4).

2.3.2 Checkpoint

Two checkpoint regions are arranged after the superblock. The start segment numbers of them are given in `Superblock.checkpoint_seg` field.

Both checkpoints have the same format. Each is encoded as a protocol buffer of the following structure:

```
message Checkpoint {
  optional fixed64 timestamp = 1;//timestamp of writing checkpoint
  optional fixed32 next_seg = 2;//next segment number
  optional InodeSimp inode_inomap = 3;//inode of inode bitmap
  optional InodeSimp inode_inovec = 4;//inode of inode vector
  optional InodeSimp inode_inotbl = 5;//inode of inode table
  repeated SegUsageRec segusage = 6;//segment usage table
}

message InodeSimp {//simple inode for special files
  optional fixed64 size = 1;//size in octets
  repeated fixed64 blkptr = 2;//block pointers [Inode::kNBlks]
  optional fixed32 ver = 3;//version
}

message SegUsageRec {//segment usage table record
  optional fixed32 nblks_live = 1;//count of live blocks
  optional fixed64 mtime = 2;//most recent data creation time
}
```

All fields are required in this version; they are marked as "optional" so that a future specification can define a different format. `next_seg` has the pointer to the next segment to be written after this checkpoint. The inodes of inode map, inode vector, inode table (Section 2.5.1) are stored in the checkpoint in a simplified format. The full content of segment usage table are also in the checkpoint as `segusage` field, which is repeated exactly N_{seg} times. In addition, a timestamp is included in the checkpoint, and is duplicated at the first 8 octets of the last block in the last segment of the checkpoint region.

The protocol buffer needs $LEN_{checkpoint} = 4{+}461{+}14{\times}N_{seg}$ octets, or $N_{checkpointblk} = \lceil LEN_{checkpoint}/LEN_{blk} \rceil$ blocks. Counting in the duplicate timestamp, a checkpoint region needs $N_{checkpointblk} + 1$ blocks, or $\lceil (N_{checkpointblk} + 1)/N_{blk} \rceil$ segments.

The two checkpoints are written alternatively. A checkpoint is valid only if the timestamp in the protocol buffer matches the timestamp in the last block. A checkpoint is preferred if it is the only valid checkpoint, or if it has a newer timestamp than the other checkpoint.

When mounting the filesystem, LFS MUST choose the preferred checkpoint; if both checkpoints are invalid, LFS MUST NOT continue mounting, and SHOULD report an error. When writing a checkpoint, LFS MUST NOT overwrite the preferred checkpoint, and MUST write to the other checkpoint region.

2.4 Log Segment

All segments starting from `Superblock.first_logseg` are used as log segments. The *log* is stored in log segments.

In each log segment, the last $N_{segsummaryblk} = \lceil (24 + 24 \times N_{blk})/LEN_{blk} \rceil$ blocks contain the segment summary, which has the metadata of the segment. Other blocks are used to store data, indirect pointers, or directory change record (Section 2.7.3).

2.4.1 Segment Summary

Segment summary is an array of records that describes a log segment and the blocks in it.

Per block metadata is stored in octets 0-$(24 \times N_{blk})$. Metadata of block i is at offset $24 \times i$, and has the following compact binary structure:

- octet 0 high 4 bits: block type, 0x0=segment summary, 0x4=directory change record, 0x5=indirect pointers, 0xF=data
- octet 0 low 4 bits, and octet 1: sequence, indicates the relative order of this block in the segment; 0x0FFF indicates this block is unused; roll-forward processes a block with sequence *seq* after processing all blocks with sequences smaller than *seq*
- octets 2-23: meaning varies for every block type, reserved if undefined

Per block metadata of a segment summary block itself contains all zeros. It is treated as a special case by LFS.

Per segment metadata is stored as 24 octets at offset $24 \times N_{blk}$, and has the following compact binary structure:

- octets 0-3: next segment number; roll-forward uses this field to find the next segment of the log
- octets 4-15: reserved
- octets 16-23: timestamp of when this segment is written; roll-forward uses this field to confirm this segment is correctly written, and stops processing if the timestamp is found to be earlier than the timestamp previous processed segment or preferred checkpoint

2.5 File

A *file* is an array of octets stored in the filesystem. Each file has an *inode* which contains the file's metadata. File content is stored in *data blocks*. Address of data blocks are represented by *block pointers*, and stored in the inode or *indirect pointer blocks*. A block pointer of zero represents a *hole* in the file; reading from a hole yields all zeros, writing to a hole makes the block pointer non-zero.

Filenames are not considered part of a file. They are considered part of the directory contains that file (Section 2.7). Each file can have 0 to 65000 names. However, a file with 0 name will cannot be accessed if it is not open, and its inode and blocks die automatically.

2.5.1 Inode

An inode contains a file's metadata, such as type, owner, permission, length, and modify time. It is stored in inode vector (Section 2.6.2) and inode table (Section 2.6.3).

The first 12 block pointers are also stored in the inode. In addition, there are one singly indirect pointer, one doubly indirect pointer, and one triply indirect pointer (Section 2.5.2). Block pointers and indirect pointers are segment-block numbers.

2.5.2 Indirect Pointer Block

An indirect pointer block is a block that contains $\lfloor LEN_{blk}/8 \rfloor$ block pointers or indirect pointers. A singly indirect pointer points to an indirect pointer block that contains block pointers. A doubly indirect pointer points to an indirect pointer block that contains singly indirect pointers. A triply indirect pointer points to an indirect pointer block that contains doubly indirect pointers.

The metadata of a data block has the following compact binary structure:

- octet 2 high 2 bits: 0 if this block contains block pointers, 1 if this block contains singly indirect pointers, 2 if this block contains doubly indirect pointers
- octets 4-7: inode number
- octets 8-11: inode version
- octets 12-15: file offset (in blocks) of the first block pointer, possibly after following a chain of indirect pointers

When an indirect pointer is updated, LFS does not update the on-disk indirect pointer block. Instead, LFS writes a new indirect pointer block, and the parent indirect pointer is updated to point to the new block. The old indirect pointer block becomes *free*. As a special case, LFS can update an indirect pointer block if it is not written to disk yet.

An indirect pointer block is *free* when any of the following is true:

- the inode (referenced by inode number field) is free
- the inode contains a different version than inode version field
- file offset is larger than file length as recorded in the inode
- the parent indirect pointer does not point to this indirect pointer block

2.5.3 Data Block

File content is sliced into LEN_{blk}-octet pieces, and each piece is stored in a data block.

The metadata of a data block has the following compact binary structure:

- octets 4-7: inode number
- octets 8-11: inode version
- octets 12-15: file offset (in blocks)
- octets 16-23: data creation time; does not change when moved by cleaner

When a file is overwritten, LFS does not update on-disk data blocks. Instead, LFS writes new data blocks, and block pointers are updated to point to the new blocks. Old data blocks become *free*. As a special case, LFS can update a data block if it is not written to disk yet.

A data block is *free* when any of the following is true:

- the inode (referenced by inode number field) is free
- the inode contains a different version than inode version field
- file offset is larger than file length as recorded in the inode
- the corresponding block pointer does not point to this data block

2.6 System Files

LFS stores certain structures in system files. Inodes 0-63 are reserved for system files.

2.6.1 Inode Bitmap

Inode bitmap is a system file that indicates whether an inode number is in-use or free. Inode bitmap file is assigned inode 16, and its inode is stored in the checkpoint.

Each inode number has one bit in inode bitmap. The bit for inode number i is at offset $\lfloor i/8 \rfloor$, bit i mod 8 (bit 0 is least significant bit, bit 7 is most significant bit). If the bit is 0, inode number i is free, otherwise inode number i is in-use.

Bits for system files (inode number 0-63) are always indicated as in-use.

2.6.2 Inode Vector

Inode vector is a system file that stores version and access time fields of inodes. Inode vector file is assigned inode 17, and its inode is stored in the checkpoint.

Each inode has 16 octets in inode vector. Inode i's fields are at offset $16 \times i$, and has the following compact binary structure:

- octets 0-3: version
- octets 4-7: reserved
- octets 8-15: access datetime

Inode vector has a much smaller length than inode table, so reading and writing inode vector is more efficient than reading and writing inode table. The version field is frequently retrieved by LFS cleaner to compare with that in segment summary, and when they don't match, LFS cleaner can conclude the block is free without reading from inode table. The access datetime field is frequently updated by filesystem callers, so storing it in inode vector saves writing to inode table [4].

2.6.3 Inode Table

Inode table is a system file that stores all other fields of inodes. Inode table file is assigned inode 18, and its inode is stored in the checkpoint.

Each inode has 256 octets in inode table. Inode i's fields are at offset $256 \times i$, and is encoded as a protocol buffer of the following structure:

```
message InodeTblEntry {
  optional uint32 mode = 1;//file type and permission
  optional uint32 nlink = 2;//number of filenames associated with this inode
  optional fixed32 uid = 3;//owner user id
  optional fixed32 gid = 4;//owner group id
  optional uint64 size = 5;//file length in octets
  optional uint32 blocks = 6;//count of data blocks allocated to this file
  optional fixed64 mtime = 7;//modify datetime
  optional fixed64 ctime = 8;//status change datetime
  repeated fixed64 blkptr = 9;//12 block pointers, one singly indirect pointer,
                  //one doubly indirect pointer, one triply indirect pointer
  optional bytes body = 10;//file body if fits
}
```

All fields except `body` are required in this version; they are marked as "optional" so that a future specification can define a different format. `body` MAY be used to store the file body if it is no more than 40 octets; `blkptr` is ignored if `body` is present.

`mode` is 16-bit unsigned integer that represents file type and permission:

- `mode & 0170000` determines file type: 0040000=directory, 0100000=regular file, 0120000=symbolic link; other values are not supported
- `mode & 0007777` represents file permission, as described in chmod(2) manpage [5]

2.7 Directory

A directory is represented as a file whose `mode & 0170000` equals 0040000. A directory file contains a list of inode numbers and corresponding file names. It is represented as a protocol buffer of the following structure:

```
message DirFile {
  repeated DirFileRec list = 1;
}

message DirFileRec {
  optional uint32 ino = 1;
  optional string name = 2;
}
```

2.7.1 Root Directory

The root directory is assigned inode 2.

2.7.2 Unlinked Directory

The unlinked directory contains all files that are `unlink`ed, but are still open. It is assigned inode 20. A file SHOULD have a name of the hexadecimal representation of its inode number. Files SHOULD be deleted when they are no longer open, or when the filesystem is being mounted.

2.7.3 Directory Change Record

A directory change record is the journal of a directory operation. It is represented as a protocol buffer of the following structure:

```
message DirChRec {
  optional uint32 ino = 1;
  optional string name = 2;
  enum Operation {
    NONE = 0;
    ADD = 1;
    REMOVE = 2;
  }
  optional Operation op = 3;
}
```

A directory change record MUST appear in a block before the new inode (with updated `nlink` field), and MUST appear before the inode of new directory file (with one or more files added and/or removed).

One or more directory change records can be arranged into a block; remaining space in that block are filled with at least one zero. A directory change record cannot cross block boundary.

The metadata of a directory change record block does not define any additional structure.

2.8 Special Files

The only special file type supported is symbolic link. Device, pipe, and socket are not supported, therefore `mknod` of these types SHOULD return an error.

2.8.1 Symbolic Link

A *symbolic link* is a special file that contains a reference to another file or directory in the form of an absolute or relative path. A symbolic link is represented as a file whose `mode & 0170000` equals 0120000.

The data of this file is the target path. It is efficiently stored in the inode table if it's within 64 octets; otherwise, one or more data blocks are needed.

3 Implementation

LFS is implemented in C++. The software project contains 10 modules. Figure 1 shows major components and their relationship in normal operation.

The following subsections describe each module.

3.1 disk module

Disk class defines an abstract disk interface that supports `Seek`, `Read`, and `Write` operations. The default implementation is a wrapper of the network disk interface. An alternative implementation provides a memory-backed disk which is useful in unit tests.

There is also a logging disk wrapper that writes a message about each operation for debugging, but it is never used.

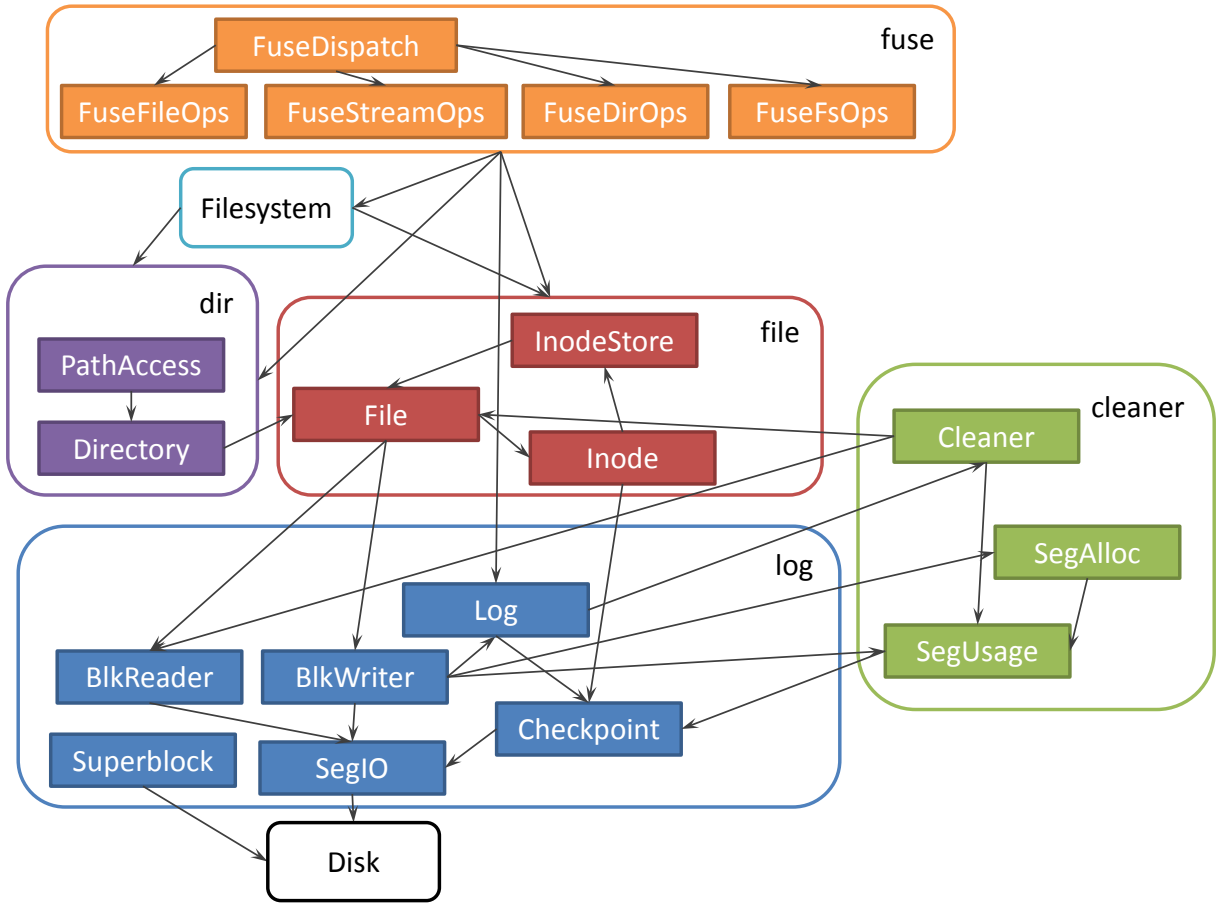

Figure 1: major components

3.2 log module

The log module is the largest and most complex component in the project.

Superblock class locates and reads the superblock from the disk on disk mount. Most other parts of the log module relies on static information in the superblock.

SegIO class provides segment abstraction above the disk. It exposes `Read` and `Write` on a segment or an octet range in a segment. Reads may be satisfied by a LRU cache, writes go directly to the disk.

On disk mount, **Checkpoints** class reads the two checkpoints from the disk, and decides which is the preferred one. The same class will also writes a checkpoint to the suitable slot when needed.

BlkWriter class allows `Put` and `Die` of a block. In `Put`, the address of a new block is decided by BlkWriter, not the caller. The `Die` operation makes a block available for reuse, and updates relevant records in the segment usage table. Once a segment is full, it is written to the disk through SegIO class; this writing can also be triggered by Log class when it needs to write a checkpoint.

BlkReader class provides `Read` of a block or an octet range in a block. It's not just a wrapper of SegIO, because some blocks may reside in the buffer of BlkWriter and should be read from there.

Log class collects above pieces together. In addition, it provides transaction support, checkpoint scheduling, and cleaner scheduling. A checkpoint is written after writing a configuration number of segments, but this may be postponed until the current transaction finishes. When the remaining cleaner segment drops

below a threshold, the cleaner is invoked immediately after writing a checkpoint.

There are also functions that manages segment summary records, and a **LogCreator** class that initializes a log on an empty disk.

3.3 cleaner module

The cleaner module works closely with the log module to keep track of free space on the disk.

SegUsage class updates the segment usage table according to calls from BlkWriter's Put and Die operations. It can calculate a priority queue of dirty segments according to their clean scores.

SegAllocator class allocates clean segments for BlkWriter to use. The default implementation uses information from segment usage table. To ensure crash consistency, only segments that were clean before the last checkpoint can be allocated. An alternative implementation allocates segments sequentially.

Cleaner class implements the actual cleaning procedure. Cleaning procedure is invoked by Log class, and is bounded as one transaction (which means no checkpoint can be written during cleaning, and the cleaned segments cannot be reused right away). An optimal batch size (number of segments to clean at a time) is decided with consideration of remaining clean segments, SegIO's cache size, etc. A few segments at a time, the data blocks and indirect pointers blocks are collected into a priority queue ordered by their age. For each block, the inode number and version in its metadata is compared to the inode bitmap and inode vector, and the block is discarded if they don't match; otherwise, File class is instructed to move the block to elsewhere (Put to BlkWriter and Die the old block) if that block is still live (according to direct or indirect pointer of that offset). The segment usage table is not directly updated, because BlkWriter will do that when called by File class. The cleaning procedure succeeds when requested number of clean segments are made, and fails when every dirty segment has been processed once but cannot make progress, or there is no more clean segment to move live blocks into.

3.4 file module

The file module provides file abstraction on the log.

Inode class represents an inode, but does not define how it is stored. **File** class takes an in-memory inode, and provides `Read`, `Write`, and `Truncate` operations. Direct and singly indirect pointers can be used; doubly and triply indirect pointers are not implemented.

InodeStore class manages the storage of inodes. It supports `AllocateIno` (allocate unused inode number), `Read`, `Write`, `Delete`, and several other operations.

There is also **SystemFileCreator** class that creates system files (root directory and inode storage files) on an empty log.

3.5 dir module

The dir module implement directories, as a file type.

Directory class supports `Get`, `Add`, `Remove` an directory entry (file name), and `List` all entries contained within. The default implementation stores directory entries in a file. An alternative implementation exposes the inode bitmap as a single root directory.

PathAccess class evaluates a pathname, walks down the directory hierarchy, and finds the inode corresponding to this pathname (or reports its absence).

3.6 fs module

Filesystem class maintains consistency between concurrent opened files. It ensures only one instance of Inode, File, or Directory object exists, even if they are opened multiple times. Therefore, every client will see the same contents. `GetInode`, `GetFile`, or `GetDirectory` method retrieves an object. The retrieved object must be returned exactly once with `Release*` method, which saves the inodes if changed, and truncates the file after nlink becomes zero and all clients close the file.

FsVerifier class verifies the correctness of an existing filesystem.

3.7 fuse module

The fuse module interfaces with FUSE to expose the filesystem to linux kernel. The features are partitioned into four classes **FuseFileOps**, **FuseStreamOps**, **FuseDirOps**, and **FuseFsOps**. Symbolic links are implemented in this module as a file type.

Functions in file, dir, and fs modules are used. In case of directory entry mutation, the transaction feature in Log class is used to prevent writing a checkpoint in the middle. Currently disk space utilization cannot be accurately calculated, but a cleaning failure is an indication of disk space shortage, so `create`, `write`, and `mkdir` operations will be rejected until the next cleaning succeeds.

FUSE C headers are used instead of C++ bindings, because none of the C++ bindings works in my environment. Therefore, some C code is necessary to initialize FUSE C library.

3.8 command module

The command module has a `main` function that parses command line arguments and starts corresponding feature. The same binary is used for `lfs`, `mklfs`, and `lfsck` commands.

3.9 Other modules

The util module provides several helper features: an LRU cache, a console logging mechanism, date representation, etc.

The exttool module collects several external code pieces such as the network-disk, and the Google C++ Testing Framework [6].

4 Testing

4.1 Unit Test

Unit tests are important to ensure quality of a software project. The disk, log, file, and dir modules have unit tests developed with Google C++ Test Framework [6]. The other modules do not have unit tests because the cost of writing unit tests is too high without using a mock framework, which I don't have experience with.

4.2 Functionality Test

Functionality test is done manually. The basic features are extensively tested and are guaranteed to work properly.

Known bugs are:

- Filesystem fails to mount if the disk has 650,000 or more sectors.
- Segment usage table may contain incorrect records after repeated cleaning failures.
- Directory operation fails when size of directory file exceeds the file length addressable by direct pointers and singly indirect pointer.
- If system crashes when an unlinked file (nlink==0) is still open, the file content is not deleted during the next mount, and the space occupied by this file is permanently lost.

4.3 Performance Test

The tests below are conducted on CS lab machine `pug.cs.arizona.edu`. The machine has one Intel Xeon X3470 CPU and 8GB of memory. The machine runs Ubuntu 12.04 64-bit with kernel 3.2.0-33.

LFS code is on local disk, and is compiled with gcc 4.6.3. LFS virtual disk is stored in a memory-mapped disk `/run/shm`. Unless otherwise noted, LFS is mounted with `./lfs -s16 -i16 -c32 -C40 /run/shm/lfs.disk /scratch/lfs`, and a new 25MB disk created with `./mklfs -s50000 -l65536 -b8 /run/shm/lfs.disk` is used for each test.

Each test is repeated 3 times, and the mean execution time is reported. Unless otherwise noted, `time` is used to measure the execution time.

mkfs test case creates a 320MB disk, and constructs a filesystem on it.

```
# lfs
./mklfs -l65536 -s50000 -b8 /run/shm/lfs.disk
# ext4
dd if=/dev/zero of=/run/shm/ext4.disk bs=512 count=50000
/sbin/mke2fs -t ext4 -b 4096 -F /run/shm/ext4.disk
```

touch+ls test case creates 100 empty files, and `ls` after each creation.

```
for i in ‘seq 1 100‘; do touch $D/$i; ls $D > /dev/null; done
```

echo+cat test case creates 100 small (16B) files, and read the file just created.

```
for i in ‘seq 1 100‘; do echo ’0123456789ABCDEF’ >$D/$i; cat $D/$i; done
```

dd test case creates 16 large (1MB) files.

```
for i in ‘seq 1 16‘; do dd if=/dev/zero of=$D/$i bs=1024 count=1024; done
```

postmark test case executes postmark-1.53 [7] benchmark. Time is reported by postmark. LFS's cleaning procedure is invoked twice during the benchmark.

```
Current configuration is:
The base number of files is 500
Transactions: 500
Files range between 500 bytes and 9.77 kilobytes in size
Working directory:
/scratch/lfs (weight=1)
100 subdirectories will be used
Block sizes are: read=512 bytes, write=512 bytes
Biases are: read/append=5, create/delete=5
```

```
Using Unix buffered file I/O
Random number generator seed is 42
Report format is verbose.
```

Table 1: performance test results

test case	LFS	memdisk	ext4	NFS
mkfs	0.169s	N/A	0.058s	N/A
touch+ls	0.536s	0.353s	0.368s	3.694s
echo+cat	0.515s	0.139s	0.140s	4.278s
dd	7.987s	0.072s	0.102s	1.681s
postmark:total	41s	1s	1s	51s
postmark:transaction	26s	1s	1s	20s

Table 1 shows the performance test results. LFS performs better than NFS, but is slower than local filesystems.

A Code Statistics

The software project contains 6162 lines of code. 59.6% is C++, 40.3% is C (or C++ header), 0.1% is other language.

Figure 2 shows the growth of code during the project. Table 2 shows contribution of individual members.

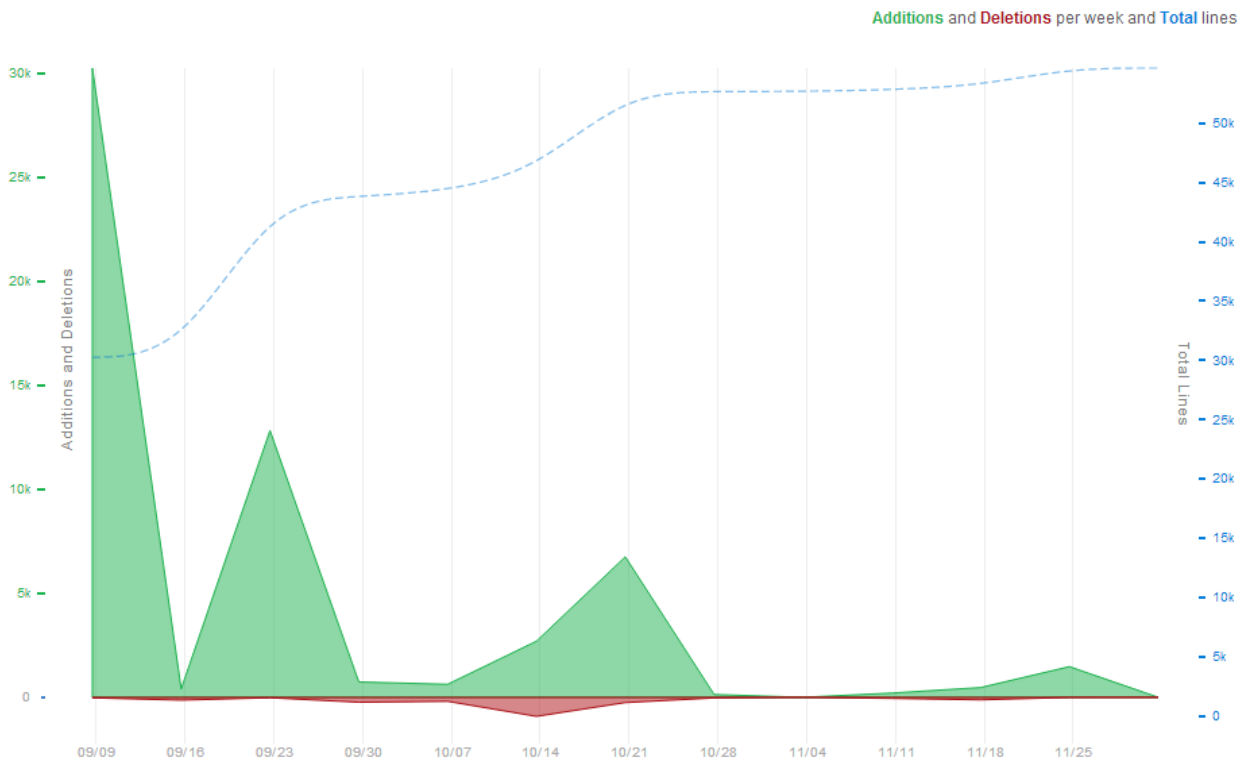

Figure 2: code frequency

Table 2: member contribution

user	git-blame lines
Junxiao	6139
Karan	23

References

[1] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *SIGOPS Oper. Syst. Rev.*, vol. 25, no. 5, pp. 1–15, Sep. 1991. [Online]. Available: http://doi.acm.org/10.1145/121133.121137

[2] "Filesystem in userspace." [Online]. Available: http://fuse.sourceforge.net/

[3] "Protocol buffers." [Online]. Available: https://developers.google.com/protocol-buffers/

[4] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin, "An implementation of a log-structured file system for unix," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX'93. Berkeley, CA, USA: USENIX Association, 1993, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267303.1267306

[5] "chmod(2) - linux man page." [Online]. Available: http://linux.die.net/man/2/chmod

[6] "Google c++ testing framework." [Online]. Available: https://code.google.com/p/googletest/

[7] J. Katcher, "Postmark: A new file system benchmark," Tech. Rep.