

## LFS

Phase 1 Due: October 27, 2009 at midnight

Phase 2 Due: December 8, 2009 at midnight

### 1. Overview

This semester you will implement a log-structured file system (LFS) that allows applications to access files and directories stored in a log. The functionality is similar to that described in the LFS paper, although I've simplified it somewhat.

This project will be done in two phases and in groups of two.

### 2. Architecture

You will implement your LFS as a process that reads and writes the disk in response to requests made by FUSE, which in turn are caused by file accesses by application programs. I will provide you with a Disk module that reads and writes a virtual disk stored in a regular Unix file. You must use FUSE and you must use Disk, what you do between these two is up to you, although you must justify the design. I would suggest the following hierarchical architecture but you are welcome to develop your own. I've sketched out the basic functionality for the layers but not provided all the details.

#### 2.1 Disk

You will be provided with the Disk layer; do not make any changes. The Disk API is defined in `disk.h`, which is also provided and you should not change:

```
disk = Disk_Open(filename, flags, *sectors)
```

```
Disk_Read(disk, offset, length, buffer)
```

```
Disk_Write(disk, offset, length, buffer)
```

*filename* is the name of the virtual disk file to open, *flags* specifies the disk behavior (you may want to specify "silent" and "async" for development, but not when you turn it in), *sectors* returns the number of sectors in the disk, *offset* is the starting offset of the I/O in sectors, and *length* is the length of the I/O in sectors.

#### 2.2 Log

The log layer is responsible for creating the log that is stored on the disk.

```
Log_Read(segment, offset, length, buffer)
```

```
Log_Write(*segment, *offset, length, buffer)
```

```
Log_Free(segment, offset, length)
```

*segment* specifies the segment number to read or free, or returns the segment number written in the case of writes. *offset* is the starting offset of the block(s) to

read/write/free, in blocks, and *length* is the number of blocks to read/write/free

### 2.2.1 Log Format

The log consists of fixed-size segments and each segment consists of fixed-size blocks. You should reserve one or more segments at the beginning of the disk to hold per-file-system metadata such as segment size, block size, number of segments, segment usage table, checkpoint regions, etc. Similarly, you should reserve one or more blocks at the beginning of each segment to hold per-segment metadata, e.g. the segment summary that contains information about the blocks in the segment.

You must write a utility *mklfs* that creates an empty LFS. It is responsible for initializing all of the on-disk data structures, metadata and creating the root directory and the ifile.

You must also write a utility *fscklfs* that scans an LFS and reports any errors such as files that do not have directory entries, directory entries that point to unused inodes, wrong segment summary information etc. You will find this utility invaluable for debugging.

### 2.2.2 Log I/O

All I/O to the disk must be done in units of segments; that is, the smallest amount of data that LFS can read or write to the virtual disk is one segment. The file system component will want to read and write in units of file blocks (see the next section), which are smaller than segments. For reads this means that LFS may be forced to read an entire segment from the disk just to access one file block. To reduce the performance penalty, LFS must implement a segment cache. The cache is stored in memory and contains the N most recently accessed segments (specified via a command-line option).

LFS only writes to the end of the log, therefore writes are handled by keeping the log "tail" segment in memory and only writing it to disk when it is full (it should also be added to the segment cache at this time). There are a couple of issues with this. First, blocks may die after they are placed in the tail segment but before the tail segment is written to disk. You should keep track of free space created by these dead blocks and reuse it for new blocks. This means that blocks may not appear in the tail segment in the order they were written, so you will have to record this order in the segment summary if you plan to implement roll-forward. The alternative, leaving dead blocks in the tail segment, causes far too many problems.

Second, during a checkpoint it may be necessary to write the tail segment to disk before it is full. This is ok, although you must ensure that the unused space is noted as such in the segment usage table and segment summary. The next log I/O should use a new segment, leaving the unused space in the partial segment to be reclaimed by the cleaner.

## 2.3 File

The file layer is responsible for implementing the file abstraction. A file is represented by an inode containing the metadata for the file, including the file type (e.g. file or directory), size, and the disk addresses of the file's blocks. The disk addresses are stored in a UNIX-like inode structure in which the inode contains four direct pointers to the first four blocks of the file and one indirect pointer to a block of direct pointers.

File\_Init(\*inode, type)

File\_Write(\*inode, offset, length, buffer)

File\_Read(\*inode, offset, length, buffer)

File\_Free(\*inode)

*inode* is the inode of the file to be accessed, *offset* is the starting offset of the I/O in bytes, and *length* is the length of the I/O in bytes.

Note that the File layer does not store the inodes in the log, it simply uses the inodes passed to it and updates their contents as necessary.

## 2.4 Directory

The Directory layer is responsible for storing inodes and implementing the directory hierarchy.

### 2.4.1 Ifile

Inodes are stored in a special file called the *ifile*. Note that the ifile is simply a special kind of file -- do not store the inodes anywhere else. Storing the inodes in the ifile allows you to grow the number of inodes incrementally, and to access the inodes using the File routines. For example, to read an inode for a particular file the Directory layer calls File\_Read and passes it the inode for the ifile and the proper offset for the inode of the desired file. The inode for the ifile itself is a bit tricky to handle since you need the inode in order to find the ifile to get the inode. Break the circularity by storing the ifile's inode in the checkpoint (note that you use the File routines to manipulate the ifile itself by passing the ifile's inode to the File routines).

### 2.4.2 Directories

A directory is simply a special kind of file that contains an array of <name, inum> pairs. Every directory must have the standard '.' and '..' entries, and multiple links to the same file must be supported. Do not implement directories any other way.

### 2.4.3 Special Files

You must support symbolic links, but you do not need to support other types of special files such as devices, named pipes, etc.

## 2.5 Cleaner

The cleaner is responsible for cleaning the log. Since it needs information about both the log and files it will have to use both the File and Log routines. It should use the same segment usage table and cost/benefit calculation as the LFS paper to determine which segments to clean. A segment is cleaned by copying its live data to the end of the log. Cleaning is controlled by two parameters to the cleaner. The first is the minimum number of free segments before cleaning begins. The cleaner starts cleaning when the number of

segments falls to this level. The second is the number of free segments at which point the cleaner stops cleaning. Note that if the disk is full the cleaner may not be able to produce the required number of free segments and will simply run forever. You can handle this by having the I/O return "disk full" if the cleaner cannot make progress because the disk is full.

You may make two assumptions to simplify cleaning. First, you can check the number of free segments before performing an operation (e.g. writing to a file) and decide whether or not to clean. You don't need to clean during an operation. Second, cleaning can be synchronous. You don't need to clean while simultaneously handling file system operations.

### 3. Crash Recovery

Your system should recover from a crash by periodically checkpointing its state. After a crash the system resumes as of the most recent checkpoint. The checkpoint interval is configurable and is the number of segments written between checkpoints. For example, a checkpoint interval of 1 means that a checkpoint will occur after every segment is written. To simplify things, it is ok to defer a checkpoint until the current write completes, even if that will exceed the current checkpoint interval. For example, if the interval is 1 but the current write spans 10 segments, it is ok to complete the write then do the checkpoint. You should not start any new writes until the checkpoint completes, however.

Checkpoints are written to a special location on disk. You'll need two checkpoints in case you crash while writing a checkpoint. The checkpoint should be very small, containing little more than the ifile inode.

### 4. FUSE

Your LFS consists of a user-level process that makes use of FUSE to service filesystem operations made by unmodified application programs. Your LFS process need not be multi-threaded. You must implement at least the following FUSE routines: *getattr*, *readlink*, *mkdir*, *unlink*, *rmdir*, *open*, *read*, *write*, *statfs*, *release*, *opendir*, *readdir*, *releasdir*, *init*, *destroy*, and *create*.

Your LFS process has the following command-line syntax:

```
lfs [options] file mountpoint
```

Where the options consist of:

```
-s num, --cache=num
```

Size of the cache in the Log layer, in segments.

```
-i num, --interval=num
```

Checkpoint interval, in segments.

```
-c num, --start=num
```

Threshold at which cleaning starts, in segments. Default is 4.

-C num, --stop=num

Threshold at which cleaning stops, in segments. Default is 8.

The *file* argument specifies the name of the virtual disk file, and *mountpoint* specifies where the LFS filesystem should be mounted. LFS must pass the '-f' argument to fuse\_main so that the process runs in the foreground instead of detaching and running in the background.

## 5. Utilities

### 5.1 mklfs

You must write a command to create and format a disk for LFS. It should zero-out the disk, then initialize all your on-disk data structures so that that your LFS process can access it properly. The initial filesystem should be empty, consisting of only the root directory and the '.' and '..' entries. The *mklfs* command has the following syntax:

**mklfs** [options] *file*

where *file* is the name of the virtual disk file to create and the options consist of:

-l *size*, --segment=*size*

Segment size. The default is 32KB.

-s *sectors*, --sectors=*sectors*

Size of the disk, in sectors. The default is 1024.

-b *size*, --block=*size*

Size of a block, in sectors. The default is 2 (1KB).

The *mklfs* command should exit with a return status of '0' if there are no errors, '1' otherwise. The *mklfs* command need not use the Disk library if you prefer.

### 5.2 fscklfs

You must write a command that checks an LFS for consistency. The *fscklfs* command has the following syntax:

**fscklfs** *file*

where *file* is the name of the virtual disk file to check. Note that *fscklfs* doesn't actually fix any errors, it simply reports them.

## 6. Turnin

### 6.1 Phase 1

Write *mklfs*, *fscklfs*, prototypes of the Log and File layers, and a stub Directory layer. The Log layer need not do cleaning. The File layer should implement direct blocks. The

Directory layer should implement a flat namespace (the only directory is the root directory). Files within the root directory are named by their inum, e.g. “/21” is the name of the file with inum 21.

## 6.2 Phase 2

Implement the remainder of the assignment.

## 6.3 Logistics

This project will be done in teams of no more than two. Since working in groups means that there is a danger of one person not carrying his or her load, I'm likely to quiz each group member orally on any part of the system during the final demos. Each group member must be familiar with the overall design and structure of their group's project.

Turn in your phases using the proper D2L dropbox. You must include a design document (PDF preferred) for both phases. Be sure to include what does and doesn't work, as well as any cool features you implemented. You will demo the project to me at the end of the semester. Your assignment will be graded on *lectura*, so verify that it works there before you turn it in (try this out early -- there are always last-minute glitches).

## 7. Extra Credit

Implement roll-forward. Roll-forward requires that you store additional information in the log so that the metadata can be updated properly. For example, a file block may have information that indicates to which file it belongs so its inode can be updated. One of the difficulties in supporting rollforward is maintaining consistency between directory entries and the inodes to which they refer. The LFS paper describes a directory log that serves this purpose. As an alternative you can add two fields, *action* and *links*, to each directory entry to support directory logging. Instead of writing out a separate directory log entry, write the directory block itself before the inode to which it refers. Use the *action* field in the entry to indicate the action that occurred, and the *links* field to indicate the new number of links to the file. Unchanged entries use a “nop” action and have an invalid *links* field.