CSc 552, Fall 2004 John H. Hartman

Project 2: Log-Structured File System

Due: December 7, 2004 at midnight

Overview

Implement a log-structured file system that allows applications to read and write files stored in a log. The functionality is similar to that described in the LFS paper, although I've simplified it a bit.

Log Format

The log is stored on a virtual disk represented by a single UNIX file. You must write a utility that creates and initializes the disk. The parameters to this utility should include the size and number of segments in the disk, and the size of the file blocks. The utility uses this information to initialize whatever metadata the disk contains.

Log I/O

All I/O to the disk is done in units of segments; that is, the smallest amount of data that can be read or written is one segment. The file system will read and write in units of file blocks (see the next section), which are smaller than segments. Therefore, reading a file block may require reading an entire segment from the disk and discarding the unneeded portions. Writing is a bit more difficult, but one way to do is it to maintain a "current segment" that is currently being filled. Space is allocated from the current segment to hold file blocks and file metadata; when the segment is full it is written to the disk and a new current segment started. One issue is that data may die before the segment is written to disk. You need not worry about writing dead data to disk, but you must ensure that this space is eventually reclaimed by the cleaner.

Files

You must implement a UNIX-like hierarchical file system. A file is a collection of fixed-size blocks as specified when the file system was created. A file is represented by an inode containing the file's metadata including 4 direct pointers to file blocks and 1 indirect pointer that points to a block containing direct pointers. Inodes are stored in a special file called the imap. Note that the imap is simply a special kind of file.

You should use the standard UNIX directory mechanism to implement file names, i.e. a directory is a file containing a list of (component,inode #) pairs. You may assume that each component of a name is at most 8 characters, and the entire pathname is at most MAXPATHLEN characters.

Cleaner

The cleaner is responsible for cleaning the log. 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 must happen concurrently with application file access (you can't stop the world while cleaning). Cleaning is controlled by two parameters to the cleaner. The first is the minumum number of free segments before cleaning begins. The cleaner starts cleaning when the number of segments falls to this number or below. The second is the number of free segments at which point the cleaner stops cleaning.

Crash Recovery

During a crash the current segment and data not yet written to the disk are lost. Your system should recover from a crash by periodically checkpointing its state and rolling forward from the most recent checkpoint. 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 the metadata can be updated. Checkpoints prevent having to rollforward from the beginning of the log. The checkpoint interval is configurable and is the number of segments that can be written before a checkpoint must be done. For example, a checkpoint interval of 1 means that a checkpoint will occur after every segment written.

Checkpoints can be written to a special location on disk as per the LFS paper, or in the log itself as per Zebra. You can put them in the log if you put checkpoint information in the segment header indicating whether or not a particular segment contains a checkpoint and where it is.

Use segment checksums to detect corruption caused by a crash when writing a segment.

Concurrency

Multiple applications may access your LFS simultaneously. You must provide some sort of synchronization to prevent metadata corruption. I think the simplest solution is to have an LFS process that all applications communicate with to access the LFS. The LFS process may be multithreaded to support cleaning and simultaneous accesses, but all LFS data structures are confined to a single process. It also makes it easy to simulate a crash by killing this process. Alternatively, you can used shared memory to share the LFS data structures between the application programs. The choice is yours, but if you go this latter route you need to provide a mechanism that simulates a crash.

API

Applications use the following routines to access your file system.

```
int
LFS_Read(
      char              *name,      // name of file read
      int               offset,     // offset at which to read
      int               size,       // amount to read
      void              *buffer);   // buffer in which to put data
```

Description

Reads *size* bytes of data from file *name* starting at offset *offset*. The data are put into *buffer*. It is not an error to read beyond the end of the file; the read is simply truncated at the end of the file and the number of bytes read returned.

Returns

- number of bytes read on success
- -1 name does not exist
- -2 name is too long
- -3 offset must be positive
- -4 size must be positive

```
int
LFS_Write(
      char              *name,      // name of file write
      int               offset,     // offset at which to write
      int               size,       // amount to write
      void              *buffer);   // data to write
```

- Description
  - Writes *size* bytes of data to file *name* starting at offset *offset*. The data written from *buffer*. The file is created if it does not already exist.
- Returns
  - number of bytes written on success
  - -2 name is too long
  - -3 offset must be positive
  - -4 size must be positive
  - -5 name is a directory
  - -6 invalid path (path to name either doesn't exist or contains a file)
  - -10 no space (no free space, inodes, directory entries, etc.)

```
int
LFS_Mkdir(
      char              *name);     // name of file write
```

- Description
  - Creates a directory with the given name.
- Returns
  - 0 success
  - -2 name is too long
  - -6 invalid path (path to name either doesn't exist or contains a file)
  - -7 name already exists
  - -10 no space (no free space, inodes, directory entries, etc.)

```
int
LFS_Rmdir(
      char              *name);     // name of file write
```

- Description
  - Deletes an empty directory with the given name.
- Returns
  - 0 success
  - -1 name does not exist
  - -2 name is too long
  - -6 invalid path (path to name either doesn't exist or contains a file)
  - -8 name is a file
  - -9 directory is not empty

```
int
LFS_Link(
      char              *old,       // old file name
      char              *new);      // new file name
```

Description
Creates a new name for a file.
Returns
0 success
-1 old does not exist
-2 old or new is too long
-5 old is a directory
-6 invalid path (path to old or new either doesn’t exist or contains a file)
-7 new already exists
-10 no space (no free space, inodes, directory entries, etc.)

```
int
LFS_Unlink(
      char              *name);     // file name to remove
```

Description
Removes the name of a file. The file is deleted if it has no more names.
Returns
0 success
-1 name does not exist
-2 name is too long
-5 name is a directory
-6 invalid path (path to name either doesn’t exist or contains a file)

Logistics

This project will be done in teams of size one or two. Since working in groups means that there is a danger of one person not carrying his or her load, I’m likely 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*. I encourage you to work in groups of two; should you decide to work alone you will not be expected to complete as much of the project as the larger groups. I leave it up to my discretion as to how much this will be, although you can expect it to be more than half of what the larger groups must complete.

Turn in your assignment using the assignment name *552lfs*. You must include a design document (PDF preferred). You will demo your project to me as part of the grading and I will read your design document prior to the demo. Be sure to include what does and doesn’t work, as well as any cool features you implemented. The demos will take place on the December 8 and 9. *Your assigment will be graded on lectura*.