Introduction

Writing to Disk Sequentially

Sample content Page 2

Writing Sequentially and Effectively

Sample content New Page

How Much To Buffer?

So, how many updates should LFS buffer before writing to disk? It depends on the disk and how high the positioning overhead is in comparison to the transfer rate.
If before each write, positioning (rotation and search over-heads) takes Tposition seconds an d the disk transfer rate is Rpeak MB/s, how much should LFS buffer before writing?

You can think of it like paying a fixed overhead of the positioning cost every time you write. So, how much do you need to write to recoup that cost? And the more you write, the better (and closer you go to peak bandwidth).
Assume we are programming D MB to get a tangible answer.
The time to write out this piece of data (Twrite) is Tposition plus D (D), or:

$T_{write} = T_{position} + D/R_{peak}$

So the effective rate of writing ($R_{effective}$), which is the amount of data written (D) divided by the total time to write it, is:

INSERT EQUATION

We want the effective rate (Reffective) close to the peak rate. We want the effective rate to be a fraction F of the peak rate, where 0<F <1. (a typical F might be 0.9, or 90% of the peak rate).

If we solve for D, we get:

INSERT EQUATIONS

For example, if a disk has a:

- Positioning Time = 10 milliseconds
- Peak Transfer Rate = 100 MB/s

and we want and effective bandwidth of 90%:

- effective bandwidth = 0.9

Then:

INSERT EQUATION

Test different numbers to discover how much buffering is required to get peak bandwidth. How much is needed to hit 95% peak? 99%?

Finding Inodes

Let us first study how to find an inode in a normal UNIX file system. A conventional file system like FFS or even the earlier UNIX file system has inodes structured in an array and placed in defined positions on disk.

For example, the original UNIX file system kept all inodes in one place. To identify an inode given an inode number and a start address, simply multiply the inode number by the inode size and add that to the start address of the on-disk array.

Finding an inode using an inode number is significantly more difficult in FFS since it divides the inode table into pieces and inserts them within each cylinder group. So one must know the size of each inode chunk and their start addresses. The rest is simple math.

Life is harder in LFS. Why? We've spread the inodes around the disk! Worse, we never overwrite in place, so the desired inode keeps shifting.

The Inode Map

To solve this, LFS designers created an inode map data structure that acts as a layer of indirection between inode numbers and inodes (imap). This structure takes an inode number as input and returns the most recent version's disk address. As a result, it's commonly implemented as a 4-byte array (a disk pointer). When an inode is written to disk, the imap is updated.

The imap must be persistent (written to disk) in order for LFS to retain track of inode positions across crashes. So, where should imap be stored on disk?

Of course, it may be fixed to the disk. Unfortunately, because it is often updated, this results in slower performance (i.e., there would be more disk seeks, between each update and the fixed location of the imap).

Instead, LFS writes parts of the inode map immediately close to where it writes new data. By doing so, LFS actually writes the new data block together with its inode and a section of it's map to the disk like follows:

INSERT IMAGE

In this image, the imap array stored in the imap block notifies LFS that the inode k is at disk address A1 and that its data block D is at address A0.

The Checkpoint Region

You may have noticed an issue. How to find the inode map now that it's fragmented over the disk? Finally, there is no magic: a file lookup requires a fixed and known location on disk.

The check-point area in LFS is just such a fixed place on disk (CR). To find the latest inode map parts, first read the checkpoint region (CR). This is because the checkpoint region is only updated regularly (every 30 seconds or so). A checkpoint area (pointing to the newest portions of the inode map) contains addresses of inodes, which lead to files (and directories) like ordinary UNIX file systems.

Here's an example of a checkpoint area (at address 0), an imap chunk, inode, and data block. A true file system would have a considerably larger CR (really two, as we'll see), much more imap pieces, and many more inodes, data blocks, etc.

INSERT IMAGE

Review: Reading a File From Disk

Let's walk through reading a file from disk, assuming we start with no memory.

1. The checkpoint area is the first on-disk data structure we have to read. It contains pointers to the entire inode map, so the LFS reads it and caches it in memory.
2. The inode number of a file is read in by the LFS, which looks up the inode number to disk address mapping in the imap, and reads the most recent version.
3. To read a block from the file, LFS uses direct, indirect, or doubly-indirect pointers, as needed.

Typically, LFS should perform the same amount of I/O as a typical file system. The entire imap is cached, so the extra work LFS does during a read is to search for the inode's address in the imap.

Directory Access

So far, we've only discussed inodes and data blocks. Some directories must be opened as well to access a file in a file system (like /home/remzi/foo). So how is directory data stored in LFS?

Lucky for us, directory structure is very much the same as in ancient UNIX file systems, with (name, inode number) mappings. LFS must, for example, write a new inode, some data, as well as the file's directory data and inode when creating a file. Remember that LFS does so consecutively (after buffering the updates for some time). This would result in the following new structures on disk:

INSERT IMAGE

The inode map provides information about the directory file dir and the newly generated file f. In this example, you would first search in the inode map (typically cached in RAM) for the directory inode (A3), then read the directory inode to discover the directory data (A2), and read this data block to get the name-to-inode-number mapping of file foo (foo, k). Then you look up inode k (A1) on the inode map, and finally read the desired data block at address A0.

The inode map also overcomes another important LFS issue, recursive update. The issue exists in any file system that transfers updates to new locations on the disk (like LFS).

Specifically, updating an inode moves it on disk. If we hadn't been diligent, this would have necessitated updating the directory that corresponds to the file, and so on up the file system tree.

LFS avoids this issue by using an inode map. Even if an inode's location changes, the directory's imap structure remains unchanged. LFS avoids the recursive update problem by directing.

Garbage Collection

Another issue with LFS is that it keeps writing the latest version of a file (inode and data) to new locations on disk. While this improves write efficiency, it also means that LFS scatters old file structure versions throughout the disk. We sneer at the old versions.

Consider a file referred to by inode number k, which links to a single data block D0. With each new inode and data block, we update the previous block. For simplicity, we omit the imap and other structures; a new chunk of imap must be written to disk to refer to the new inode):

INSERT IMAGE

As seen in the diagram, the inode and data block have two versions on disk, one ancient (left) and one current (right) (the one on the right). By (logically) changing a data block, LFS must persist new structures, leaving old versions on the disk.

Assume we instead append a block to the original file k. The inode is generated again, but it still points to the original data block. So it's still there, and part of the current file system:

INSERT IMAGE

So, what should we do with these old inodes, data blocks, etc.? A versioning file system keeps track of the multiple versions of a file and allows users to restore prior versions (for example, when they unintentionally overwrite or delete a file).

To make up for this, LFS must periodically discover and clean old dead versions of file data, inodes, and other structures, freeing up disk blocks for later writes. Cleaning is a type of garbage collection, which is used in computer languages to automatically free up memory.

We previously mentioned segments as the mechanism that allows big writes to disk in LFS. They are, in fact, essential to successful cleaning. Consider what would happen if the LFS cleaner only cleaned single data blocks, inodes, etc. The outcome is a file system with free holes mixed in with assigned space. Write performance would suffer as LFS could not discover a big continuous region to write sequentially and quickly to disk.

As a result, the LFS cleaner frees up significant amounts of space for subsequent writing. This is how basic cleaning works. To free up space for writing, the LFS cleaner periodically reads in old (partially-used) segments, decides which blocks are live inside them, and writes out a fresh set of segments with just the live blocks within them. It should read M existing

segments, compress their content into N new segments (where N M), and write the N segments to disk in new places. The file system can then use the previous M segments for further writes.

But now we have two issues. The first is mechanism: how does LFS know which blocks in a segment are alive? In addition, the cleaner should be scheduled to clean certain segments.

Block Liveness

First, the mechanism. LFS must be able to determine if a data block D within an on-disk segment S is live. To do so, LFS adds more information to each block segment. LFS includes the inode number (which file it belongs to) and offset for each data block D. (which block of the file this is). This data is stored in the segment summary block at the segment's head.

With this information, determining a block's status is simple. Find the inode number N and offset T of block D at address A in the segment summary block. Next, locate N in imap and read it from disk (perhaps it is already in memory, which is even better). Finally, use the offset T to find the file's Tth block on disk. LFS can determine if block D is alive if it links to disk address A. If it points elsewhere, LFS knows that D is no longer needed and that this version is obsolete. The following is a pseudocode summary:

```
(N, T) = SegmentSummary[A];
    inode = Read(imap[N]);
    if (inode[T] == A)
      // block D is alive
    else
      // block D is garbage
```

The segment summary block (SS) records that the data block at address A0 is actually a portion of file k at offset 0. You may find the inode by checking the imap for k.

INSERT IMAGE

LFS uses various shortcuts to speed up the liveness determination procedure. When a file is truncated or destroyed, LFS increments its version number and updates the imap. By comparing the on-disk version number to an imap version number, LFS can prevent unnecessary reads and bypass the lengthy check outlined above.

Crash Recovery and The Log

Finally, what if the system breaks while LFS is writing? As mentioned previously in the chapter on updating, file system crashes during updates are a concern for LFS.

Normally, LFS buffers writes in a segment and then writes the segment to disk when the segment is full or a time period has passed. LFS organizes these writes in a log, where each segment points to the next segment to be written. LFS refreshes the checkpoint region. Clearly, crashes might occur during either activity (write to a segment, write to the CR). When LFS writes to these structures, it can crash.

Let's start with the second. LFS maintains two CRs, one at each end of the disk, and writes to them alternately to assure atomic CR updates. LFS also uses a rigorous protocol for updating the CR with new inode map pointers and other data; it puts out a header (with timestamp), then the body, and finally one last block (also with a timestamp). LFS can identify a system crash during a CR update by noticing an inconsistent pair of timestamps. LFS will always utilize the most current CR with constant timestamps, ensuring a consistent CR update.

Now for the first case. It's possible that the last consistent snapshot of the file system is rather old. When LFS is restarted, it will read the checkpoint region, the imap pieces it points to, and any files or directories it hasn't already read.

To improve this, LFS attempts to reconstruct many of those segments using a database approach called roll forward. From the last checkpoint section, determine the end of the log (which is included in the CR), and use that to go through the next segments, looking for valid updates. And if there are, LFS recovers much of the data and metadata written since the last checkpoint.

Summary

LFS changes the way disks are updated. Instead of overwriting files, LFS always writes to an unused area of the disk, and then cleans it later. By combining all updates into an in-memory segment and writing them sequentially, LFS can achieve great writing efficiency.

LFS's massive writes are great for performance on many devices. Large writes reduce positioning time on hard drives and eliminate the small-write problem in parity-based RAIDs like RAID-4 and RAID-5. Large I/Os are required for great performance on Flash-based SSDs, so LFS-style file systems may be an appropriate choice for these new mediums.

This method generates garbage; old copies of data are spread around the disk, and to reclaim such space for future use, one must periodically clean old segments. Cleaning became a hot topic in LFS, and concerns about cleaning costs may have limited LFS' initial impact. The conceptual legacy of LFS lives on in certain recent commercial file systems, such as NetApp WAFL, Sun ZFS, Linux btrfs, and even newer flash-based SSDs. By offering historical versions of the file system via snapshots, users may access old files anytime they unintentionally destroyed current ones.