**Chapter 43: Log-structured File Systems**

The motivation is that **the system memories are growing**, **large gap between random I/O performance and sequential I/O performance**, **Existing file systems perform poorly on many common workloads** and **File systems are not RAID-aware**.

A new file system is **LFS** or **Log-structured File System**. When writing to disk, LFS first buffer all updates (including metadata) in an in-memory **segment**. When the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk. LFS never overwrites existing data, but rather always writes segments to free locations.

**43.1 Writing To Disk Sequentially**

The first challenge is to transform all updates to file-system state into a series of sequential writes to disk.

Consider writing a data block D to a file. In the on-disk layout, D is written at disk address A0:

Shape, rectangle

Description automatically generated

However, when a user writes a data block, it is not only data that gets written to disk. There is also other **metadata** that needs to be updated. In this case, we also write the **inode** of the file to disk and have it points to block D. The data block and inode would look like this:

Shape, rectangle

Description automatically generated

This basic idea, of simply writing all updates (such as data blocks, inodes, etc.) to the disk sequentially, sits at the heart of LFS.

**43.2 Writing Sequentially And Effectively**

Writing to disk sequentially is not (alone) enough to guarantee efficient writes. Rather, we must issue a large number of contiguous writes to the drive to achieve good write performance.

Thus, LFS uses an ancient technique called write buffering. Before writing to the disk, LFS keeps track of updates in memory; when it has received a sufficient number of updates, it writes them to disk all at once, thus ensuring efficient use of the disk.

The large chunk of updates LFS writes at a time is referred to as **segment**.

For example, if we have two sets of updates where the first update is four blocks writes to file j and the other is just one block being added to file k, LFS commits the entire segment of seven blocks to disk at once:

Diagram

Description automatically generated

**43.3 How Much To Buffer?**

Assume that we are writing out D MB. The time to write out this chunk of data (Tw) is the positioning time (rotation and seek) plus the time to transfer (D / Rp) where Rp is the peak disk transfer rate:

Tw = T position + D/Rp

Thus, the effective rate of writing is

R = D/Tw

We would want the effective rate to be close to peak disk transfer rate.

We can also solve for D:

Text, letter

Description automatically generated

**43.4 Problem: Finding Inodes**

In Unix file system, all inodes are organized in an array and placed on disk at fixed locations. Given an inode number and the start address, to find a particular inode, you can calculate its exact disk address simply by multiplying the inode number by the size of an inode, and adding that to the start address of the on-disk array.

In FFS, it is complicated because it splits up the inode table into chunks and places a group of inodes within each cylinder group. given an inode number and the start address, to find a particular inode, you can calculate its exact disk address simply by multiplying the inode number by the size of an inode, and adding that to the start address of the on-disk array.

LFS is more difficult as we’ve managed to scatter the inodes all throughout the disk. In addition, we never overwrite in place, and thus the latest version of an inode.

**43.5 Solution Through Indirection: The Inode Map**

To remedy this, the designers of LFS introduced a **level of indirection** between inode numbers and the inodes through a data structure called the **inode map (imap)**. The imap is a structure that takes an inode number as input and produces the disk address of the most recent version of the inode. Thus, it would be often implemented as a simple array. Any time an inode is written to disk, the imap is updated with its new location.

The LFS places chunks of the inode map right next to where it is writing all of the other new information. Thus, when appending a data block to a file k, LFS actually writes the new data block, its inode, and a piece of the inode map all together onto the disk, as follows:

Table

Description automatically generated

the piece of the imap array stored in the block marked imap tells LFS that the inode k is at disk address A1. This inode, in turn, tells LFS that its data block D is at address A0.

**43.6 Completing The Solution: The Checkpoint Region**

To find the inode map, LFS has just such a fixed place on disk for this, known as the **checkpoint region (CR)**. The checkpoint region contains pointers to the latest pieces of the inode map, and thus the inode map pieces can be found by reading the CR first. The CR is also updated periodically, so the performance is not affected. Thus, the overall structure of the on-disk layout contains a checkpoint region which points to the latest pieces of the inode map, the inode map pieces contain addresses of the inodes and the inodes point to files.

Here is an example of the checkpoint region (note it is all the way at the beginning of the disk, at address 0), and a single imap chunk, inode, and data block:

Chart, box and whisker chart

Description automatically generated

**43.7 Reading A File From Disk: A Recap**

Assume we have nothing in memory to begin. The first on-disk data structure we must read is the checkpoint region. The checkpoint region contains pointers (i.e., disk addresses) to the entire inode map, and thus LFS then reads in the entire inode map and caches it in memory. After this point, when given an inode number of a file, LFS simply looks up the inode-number to inode-diskaddress mapping in the imap, and reads in the most recent version of the inode. To read a block from the file, at this point, LFS proceeds exactly as a typical UNIX file system, by using direct pointers or indirect pointers or doubly-indirect pointers as need be. In the common case, LFS should perform the same number of I/Os as a typical file system when reading a file from disk; the entire imap is cached and thus the extra work LFS does during a read is to look up the inode’s address in the imap.

**43.8 What About Directories?**

Directory structure is basically identical to classic UNIX file systems, in that a directory is just a collection of (name, inode number) mappings. For example, when creating a file on disk, LFS must both write a new inode, some data, as well as the directory data and its inode that refer to this file. LFS will do it sequentially on the disk. Thus, creating foo in a directory would lead to the following new structures:

Diagram

Description automatically generated

The piece of the inode map contains the information for the location of both the directory file dir as well as the newly-created file f. Thus, when we access file foo, we would first look in the inode map to find the location of the inode of directory dir (A3). Then, we read the directory node that gives us the location of the directory data (A2). Reading this gives us the name and inode number mapping of foo and k. We then consult the inode map again to find the location of inode number k (A1), and finally read the desired data block at address A0.

This is a serious problem in LFS called **recursive update problem**. The problem arises in any file system that never updates in place (such as LFS), but rather moves updates to new locations on the disk. Specifically, whenever an inode is updated, its location on disk changes. If we hadn’t been careful, this would have also entailed an update to the directory that points to this file, which then would have mandated a change to the parent of that directory, and so on, all the way up the file system tree.

LFS cleverly avoids this problem with the inode map. Even though the location of an inode may change, the change is never reflected in the directory itself; rather, the imap structure is updated while the directory holds the same name-to-inode-number mapping. Thus, through indirection, LFS avoids the recursive update problem.

**43.9 A New Problem: Garbage Collection**

LFS repeatedly writes the latest version of a file (including its inode and data) to new locations on disk. This process, while keeping writes efficient, implies that LFS leaves old versions of file structures scattered throughout the disk. We call these old versions **garbage**.

s imagine the case where we have an existing file referred to by inode number k, which points to a single data block D0. We now update that block, generating both a new inode and a new data block. The resulting on-disk layout of LFS would look something like this:

Chart, box and whisker chart

Description automatically generated

In addition, imagine we instead append a block to that original file k. In this case, a new version of the inode is generated, but the old data block is still pointed to by the inode. Thus, it is still live and very much part of the current file system:

Chart, box and whisker chart

Description automatically generated

Such file system is called versioning file system because it keeps track of the different version of a file.

LFS, however, only keeps the latest **live** version of a file. Thus, LFS must periodically find these old versions and **clean** them (make blocks on disk free again for use in subsequent writes). The process of cleaning is a form of **garbage collector**.

However, problems arise since we are dealing with segments. Freeing blocks would create a number of free **holes** mixed between allocated space on disk. Write performance would drop considerably, as LFS would not be able to find a large contiguous region to write to disk sequentially and with high performance.

LFS cleaner works on segment-by-segment basis. Thus, this clears up large chunks of space for subsequence writing. Periodically, the LFS cleaner reads in a number of old (partially-used) segments, determines which blocks are live within these segments, and then write out a new set of segments with just the live blocks within them, freeing up the old ones for writing. Specifically, we expect the cleaner to read in M existing segments, **compact** their contents into N new segments (where N < M), and then write the N segments to disk in new locations. The old M segments are then freed and can be used by the file system for subsequent writes.

**43.10 Determining Block Liveness**

Given a data block D within an on-disk segment S, LFS must be able to determine whether D is live. LFS does this by adding extra information to each segment that describes each block. This information is recorded in a structure at the head of the segment known as the **segment summary block**.

Given this information, it is straightforward to determine whether a block is live or dead. For a block D located on disk at address A, look in the segment summary block and find its inode number N and offset T. Next, look in the imap to find where N lives and read N from disk. Finally, using the offset T, look in the inode (or some indirect block) to see where the inode thinks the Tth block of this file is on disk. If it points exactly to disk address A, LFS can conclude that the block D is live. If it points anywhere else, LFS can conclude that D is not in use (i.e., it is dead) and thus know that this version is no longer needed:

Text

Description automatically generated

There are some shortcuts LFS takes to make the process of determining liveness more efficient. For example, when a file is truncated or deleted, LFS increases its version number and records the new version number in the imap. By also recording the version number in the on-disk segment, LFS can short circuit the longer check described above simply by comparing the on-disk version number with a version number in the imap, thus avoiding extra reads.

**43.11 A Policy Question: Which Blocks To Clean, And When?**

Determining when to clean is easier; either periodically, during idle time, or when you have to because the disk is full.

Determining which blocks to clean is more challenging. We will determine this by **hot** and **cold** segment. A **hot** segment is one in which the contents are being frequently over-written; thus, for such a segment, the best policy is to wait a long time before cleaning it, as more and more blocks are getting over-written (in new segments) and thus being freed for use. A **cold** segment, in contrast, may have a few dead blocks but the rest of its contents are relatively stable.

**43.12 Crash Recovery And The Log**

During normal operation, LFS buffers writes in a segment, and then (when the segment is full, or when some amount of time has elapsed), writes the segment to disk. LFS organizes these writes in a log, i.e., the checkpoint region points to a head and tail segment, and each segment points to the next segment to be written.

To ensure that the CR update happens atomically, LFS actually keeps two CRs, one at either end of the disk, and writes to them alternately. LFS also implements a careful protocol when updating the CR with the latest pointers to the inode map and other information; specifically, it first writes out a header (with timestamp), then the body of the CR, and then finally one last block (also with a timestamp). If the system crashes during a CR update, LFS can detect this by seeing an inconsistent pair of timestamps. LFS will always choose to use the most recent CR that has consistent timestamps, and thus consistent update of the CR is achieved.

Because LFS writes the CR every 30 seconds or so, the last consistent snapshot of the file system may be quite old. Thus, upon reboot, LFS can easily recover by simply reading in the checkpoint region, the imap pieces it points to, and subsequent files and directories; however, the last many seconds of updates would be lost.

To improve upon this, LFS tries to rebuild many of those segments through a technique known as **roll forward** in the database community. The basic idea is to start with the last checkpoint region, find the end of the log (which is included in the CR), and then use that to read through the next segments and see if there are any valid updates within it. If there are, LFS updates the file system accordingly and thus recovers much of the data and metadata written since the last checkpoint.