42

Log-structured File Systems

In the early 90's, a group at Berkeley led by Professor John Ousterhout and graduate student Mendel Rosenblum developed a new file system known as the log-structured file system [RO91]. Their motivation to do so was based on the following observations:

- **Memory sizes were growing**: As memory got bigger, more data could be cached in memory. As more data is cached, disk traffic would increasingly consist of writes, as reads would be serviced in the cache. Thus, file system performance would largely be determined by its performance for writes.
- **There was a large and growing gap between random I/O performance and sequential I/O performance**: Transfer bandwidth increases roughly 50%-100% every year; seek and rotational delay costs decrease much more slowly, maybe at 5%-10% per year [P98]. Thus, if one is able to use disks in a sequential manner, one gets a huge performance advantage, which grows over time.
- **Existing file systems perform poorly on many common workloads**: For example, FFS [MJLF84] would perform a large number of writes to create a new file of size one block: one for a new inode, one to update the inode bitmap, one to the directory data block that the file is in, one to the directory inode to update it, one to the new data block that is apart of the new file, and one to the data bitmap to mark the data block as allocated. Thus, although FFS would place all of these blocks within the

same block group, FFS would incur many short seeks and subsequent rotational delays and thus performance would fall far short of peak sequential bandwidth.

- **File systems were not RAID-aware**: For example, RAID-4 and RAID-5 have the **small-write problem** where a logical write to a single block causes 4 physical I/Os to take place. Existing file systems do not try to avoid this worst-case RAID writing behavior.

An ideal file system would thus focus on write performance, and try to make use of the sequential bandwidth of the disk. Further, it would perform well on common workloads that not only write out data but also update on-disk metadata structures frequently. Finally, it would work well on RAIDs as well as single disks.

The new type of file system Rosenblum and Ousterhout introduced was called **LFS**, short for the **Log-structured File System**. When writing to disk, LFS first buffers 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 (i.e., LFS never overwrites existing data, but rather *always* writes segments to a free part of the disk). Because segments are large, the disk is used quite efficiently, and thus performance of the file system approaches the peak performance of the disk.

THE CRUX:
HOW TO MAKE ALL WRITES SEQUENTIAL WRITES?

How can a file system turns all writes into sequential writes? For reads, this task is impossible, as the desired block to be read may be anywhere on disk. For writes, however, the file system always has a choice, and it is exactly this choice we hope to exploit.

42.1 Writing To The Log: Some Details

Let's try to understand this a little bit better through an example. Imagine we are appending a new block to a file; assume that the file already exists but currently has no blocks allocated to it (it is zero

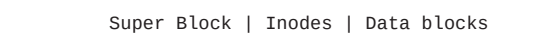
sized). To do so, LFS of course places the data block D in this in-memory segment:

However, we also must update the inode to now point to the block. Because LFS wants to make all writes sequential, it also must include the inode I in the update to disk. Thus, the segment (still in memory) now looks like this:

Note further that I is also updated to point to D (and also note that the pointer within I is a disk address, and thus when placing I in the segment, LFS must have an idea of where this segment will be written to disk). Assume this type of activity continues and the segment finally fills up and is written to disk. So far, so good. We have now written out I and D to disk, and the write to disk was efficient. Unfortunately, we have our first real problem: how can we find the inode I?

42.2 How Can We Find Those Pesky Inodes?

To understand how we find an inode in LFS, let us first make sure we understand how to find an inode in a typical UNIX file system. In a typical file system such as FFS, or even the old UNIX file system, finding inodes is really easy. They are organized in an array and placed on disk at a fixed location (or locations). For example, the old UNIX FS keeps all inodes at a fixed portion of the disk. Thus, 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. Here is what this looks like on disk:



If we expand this a bit, and assume a single block for the super block, and that we have ten blocks for inodes, we get:

```
Super Block |             Inodes              | Data blocks
    b0      | b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 | b11 ...
```

Imagine we know that each inode block of size 512 bytes, and that each inode is of size 128 bytes, and let us also assume that inodes are numbered from 0 to 39. We thus get this picture, with 4 inodes (i.e., 512/128) in each block:

```
Super Block |             Inodes              | Data blocks
    b0      | b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 | b11 ...
            |  0  4  8 12 16 20 24 28 32 36  |
            |  1  5  9 13 17 21 25 29 33 37  |
            |  2  6 10 14 18 22 26 30 34 38  |
            |  3  7 11 15 19 23 27 31 35 39  |
```

Thus, to find inode 14, we first divide 14 by 4 and get 3 (we use integer division); thus inode 14 is in the "third" block of inodes (0 is the "zeroth" block of inodes, 1 is the "first", and so on). Because the inode array starts at sector address 1, we add 3 to 1 and get sector 4 (b4); now we know which block to read to fetch inode 14. Then we do 14 mod 4 to get which inode within the block (b4) to read: 2 (again starting at 0). It is just a simple calculation.

Finding an inode given an inode number in FFS is only slightly more complicated; FFS splits up the array into chunks and places a group of inodes within each cylinder group. Thus, one must know how big each chunk of inodes is and the start addresses of each. After that, the calculations are similar and also easy.

In LFS, life is more difficult. Why? Well, we've managed to scatter the inodes all throughout the disk! Worse, we never overwrite in place, and thus the latest version of an inode (i.e., the one we want) keeps moving.

42.3 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, you can imagine it would often be implemented as a simple array, with 4 bytes (a disk pointer) per entry. Any time an inode is written to disk, the imap is updated with its new location.

DESIGN TIP: USE A LEVEL OF INDIRECTION

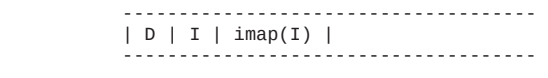
People often say that the solution to all problems in Computer Science is simply a **level of indirection**. This is clearly not true; it is just the solution to *most* problems. You certainly can think of every virtualization we have studied, e.g., virtual memory, as simply a level of indirection. And certainly the inode map in LFS is a virtualization of inode numbers. Hopefully you can see the great power of indirection in these examples, allowing us to freely move structures around (such as pages in the VM example, or inodes in LFS) without having to change every reference to them. Of course, indirection can have a downside too: **extra overhead.** So next time you have a problem, try solving it with indirection. But make sure to think about the overheads of doing so first.

42.4 Even More Problems: Where To Put The Inode Map?

The imap, unfortunately, needs to be kept persistent (i.e., written to disk); doing so allows LFS to keep track of the locations of inodes across crashes, and thus operate as desired. Thus, a question: where should the imap live?

It could live on a fixed part of the disk, of course. Unfortunately, as it gets updated frequently, this would then require the segment writes to be followed by writes to the imap, and hence performance would suffer (i.e., there would be more disk seeks, between each segment and the fixed location of the imap).

Thus, LFS places pieces of the inode map into the current segment as well. Thus, when writing a data block to disk (as above), we might actually see:



where imap(I) is the piece of the inode map that tells us where inode I is on disk. Note that imap(I) will also include the mapping information for some other inodes that are near inode I in the imap.

The clever reader might have noticed a problem here. How do we find the inode map, now that pieces of it are also now spread

across the disk? LFS finally keeps a fixed place on disk for this, and it is 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. Note the checkpoint region is only updated periodically (say every 30 seconds or so), and thus performance is not ill-affected. Thus, the overall structure of the on-disk layout contains a checkpoint region and a bunch of segments, some of which are in use, and some of which are yet to be used, as we see here:

```
    ___    __________    ___
    | |    |        |    | |  (pointer to the next segment in the log)
    | v    |        v    | v
-------------------------------------------------
| CR | seg1 | FREE | seg2 | seg3 | FREE | ....
-------------------------------------------------

CR      : contains pointers to pieces of the imap
segment : contains pieces of the imap, inodes, data, etc.
```

As you can also see from the picture, each segment also contains a pointer to the next segment; the chain of these segments thus forms the **log**, and hence the overall name of the log-structured file system. By starting at the checkpoint region, one should be able to find the first segment that was written to disk and follow a chain of segments on the disk to see what updates have taken place.

42.5 Reading A File From Disk: A Recap

To make sure you understand what is going on, let us now read a file from disk. 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-disk-address 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. Thus, 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 only extra work LFS does during file read is to look up the address of the inode in the imap.

42.6 A New Problem: Garbage Collection

You may have noticed another problem with LFS; it keeps writing newer version of a file, its inode, and in fact all data to new parts of the disk. This process, while keeping writes efficient, implies that LFS leaves older versions of a file all over the disk, scattered throughout a number of older segments.

One could keep those older versions around and allow users to restore old file versions (for example, when they accidentally overwrite or delete a file, it could be quite handy to do so); such a file system is known as a **versioning file system** because it keeps track of the different versions of a file. However, LFS instead keeps only the latest live version of a file; thus (in the background), LFS must periodically find these old dead versions of file data, inodes, etc., and **clean** them; cleaning should thus make blocks on disk free again for use in a subsequent segment write. Note that the process of cleaning is a form of **garbage collection**, a similar method that arises in languages that automatically free unused memory for programs.

The basic LFS cleaning process works as follows. Periodically, the LFS cleaner must read in a number of old (partially-used) segments, determine which blocks are live within the segment, and then write out a new set of segments with just the live blocks within them. 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. It is such cleaning that leads to free space between used segments, as shown in the picture above.

We are now left with two problems, however. The first is mechanism: how can LFS tell which blocks within a segment are live, and which are dead? The second is policy: how often should the cleaner run, and which segments should it pick to clean?

42.7 How Can We Determine Which Blocks Are Live?

We address the mechanism first. Given a data block D within an on-disk segment S, LFS must be able to determine whether D is live. To do so, LFS adds a little extra information to each segment that describes each block. Specifically, LFS includes, for each data block D, its inode number (which file it belongs to) and its offset (which block of the file this is). This information is recorded in a little 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 I and offset T. Next, look in the imap to find where I lives and read I from disk (perhaps it is already in memory, which is even better). Finally, using the offset T, look in the inode (or some indirect block) to see where the Tth block of this file is on disk. If it points exactly to disk address A, LFS can conclude that this block 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.
(a PICTURE here would be useful)

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 maintained in the imap, and thus avoid extra reads.

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

On top of the mechanism described above, LFS must build a set of policies to determine both when to clean and which blocks are worth cleaning. 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, and has been the subject of many research papers. In the original LFS paper [RO91], the authors describe an approach which tries to segregate *hot* and *cold* blocks. A hot block is one in which the contents are being frequently over-written; thus, for such a block, 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 block, in contrast, may have a few dead blocks but the rest of its contents are relatively stable. Thus, the authors conclude that one should clean cold segments sooner and hot segments later, and develop a heuristic that does exactly that. However, as with most policies, this is just one approach, and by definition is not "the best" approach; later approaches show how to do better [MR+97].

42.9 Crash Recovery

Crash recovery begins with the checkpoint region; the latest checkpoint region points to pieces of the imap, and those point to inodes representing a snapshot of file system state. However, for performance reasons, the checkpoint region is only flushed to disk periodically; thus, a crash will leave a number of segments on the disk that are not pointed to by the latest checkpoint update.

LFS tries to recover many of those segments through recovery, which takes place when you mount the file system after a crash. LFS does this by a technique known as **roll forward** in the database community. The basic idea is to start with the last checkpoint, find the end of the log, and then use that to read through the next segment and see if there are any valid updates within it. If so, update the file system accordingly and thus recover data and metadata written since the last checkpoint.

There are some more details here (tricky cases and such) which I have not yet included. Sorry!

42.10 Summary

LFS introduces a new approach to updating the disk. Instead of over-writing files in places, LFS always writes to an unused portion of the disk, and then later reclaims that old space through cleaning. This approach, which in database systems is called **shadow paging** [L77] and in file-system-speak is sometimes called **copy-on-write**, enables highly efficient writing, as LFS can gather all updates into an in-memory segment and then write them out together sequentially.

The downside to this approach is that it generates garbage; old copies of the data are scattered throughout the disk, and if one wants

to reclaim such space for subsequent usage, one must clean old segments periodically. Cleaning became the focus of much controversy in LFS, and concerns over cleaning costs [SS+95] perhaps limited LFS's initial impact on the field. However, some modern commercial file systems, including NetApp's WAFL and Sun's ZFS both adopt a similar copy-on-write approach to writing to disk, and thus the intellectual legacy of LFS lives on in these modern file systems.

References

[L77] "Physical Integrity in a Large Segmented Database"
R. Lorie
ACM Transactions on Databases, 1977, Volume 2:1, pages 91-104 *The original idea of shadow paging is presented here.*

[MJLF84] "A Fast File System for UNIX"
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry
ACM TOCS, August, 1984, Volume 2, Number 3, pages 181-197

[MR+97] "Improving the performance of log-structured file systems with adaptive methods" Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, Thomas E. Anderson
SOSP 1997, pages 238-251, October, Saint Malo, France

[P98] "Hardware Technology Trends and Database Opportunities"
David A. Patterson
ACM SIGMOD '98 Keynote Address, Presented June 3, 1998, Seattle, Washington
Available: http://www.cs.berkeley.edu/ pattrsn/talks/keynote.html

[RO91] "Design and Implementation of the Log-structured File System"
Mendel Rosenblum and John Ousterhout, SOSP '91
More information is available in the dissertation:
http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-696.pdf

[SS+95] "File system logging versus clustering: a performance comparison"
Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan
USENIX 1995 Technical Conference, New Orleans, Louisiana, 1995