43

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:

- **System memories are growing**: As memory gets bigger, more data can be cached in memory. As more data is cached, disk traffic increasingly consists of writes, as reads are serviced by the cache. Thus, file system performance is largely determined by its write performance.
- **There is a large gap between random I/O performance and sequential I/O performance**: Hard-drive transfer bandwidth has increased a great deal over the years [P98]; as more bits are packed into the surface of a drive, the bandwidth when accessing said bits increases. Seek and rotational delay costs, however, have decreased slowly; it is challenging to make cheap and small motors spin the platters faster or move the disk arm more quickly. Thus, if you are able to use disks in a sequential manner, you gain a sizeable performance advantage over approaches that cause seeks and rotations.
- **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 a part of the new file, and one to the data bitmap to mark the data block as allocated. Thus, although FFS places all of these blocks within the same block group, FFS incurs many short seeks and subsequent rotational delays and thus performance falls far short of peak sequential bandwidth.
- **File systems are not RAID-aware**: For example, both 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.

TIP: DETAILS MATTER

All interesting systems are comprised of a few general ideas and a number of details. Sometimes, when you are learning about these systems, you think to yourself "Oh, I get the general idea; the rest is just details," and you use this to only half-learn how things really work. Don't do this! Many times, the details are critical. As we'll see with LFS, the general idea is easy to understand, but to really build a working system, you have to think through *all* of the tricky cases.

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.

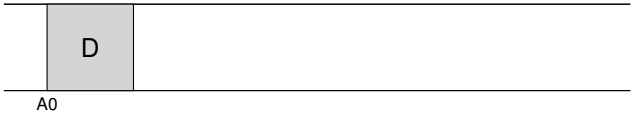
The new type of file system Rosenblum and Ousterhout introduced was called **LFS**, short for the **Log-structured File System**. When writing to disk, LFS first buffers all updates (including metadata!) in an in-memory **segment**; when the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk. LFS never overwrites existing data, but rather *always* writes segments to free locations. Because segments are large, the disk (or RAID) is used efficiently, and performance of the file system approaches its zenith.

THE CRUX:
HOW TO MAKE ALL WRITES SEQUENTIAL WRITES?

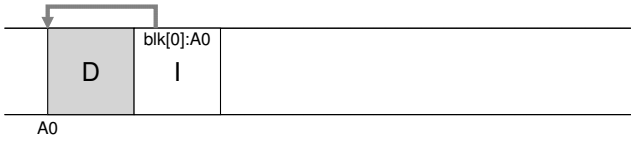
How can a file system transform all writes into sequential writes? For reads, this task is impossible, as the desired block to be read may be anywhere on disk. For writes, however, the file system always has a choice, and it is exactly this choice we hope to exploit.

43.1 Writing To Disk Sequentially

We thus have our first challenge: how do we transform all updates to file-system state into a series of sequential writes to disk? To understand this better, let's use a simple example. Imagine we are writing a data block D to a file. Writing the data block to disk might result in the following on-disk layout, with D written at disk address $A0$:



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, let's also write the **inode** (I) of the file to disk, and have it point to the data block D. When written to disk, the data block and inode would look something like this (note that the inode looks as big as the data block, which generally isn't the case; in most systems, data blocks are 4 KB in size, whereas an inode is much smaller, around 128 bytes):



This basic idea, of simply writing all updates (such as data blocks, inodes, etc.) to the disk sequentially, sits at the heart of LFS. If you understand this, you get the basic idea. But as with all complicated systems, the devil is in the details.

43.2 Writing Sequentially And Effectively

Unfortunately, writing to disk sequentially is not (alone) enough to guarantee efficient writes. For example, imagine if we wrote a single block to address A, at time T. We then wait a little while, and write to the disk at address $A + 1$ (the next block address in sequential order), but at time $T + \delta$. In-between the first and second writes, unfortunately, the disk has rotated; when you issue the second write, it will thus wait for most of a rotation before being committed (specifically, if the rotation takes time $T_{rotation}$, the disk will wait $T_{rotation} - \delta$ before it can commit the second write to the disk surface). And thus you can hopefully see that simply writing to disk in sequential order is not enough to achieve peak performance; rather, you must issue a large number of *contiguous* writes (or one large write) to the drive in order to achieve good write performance.

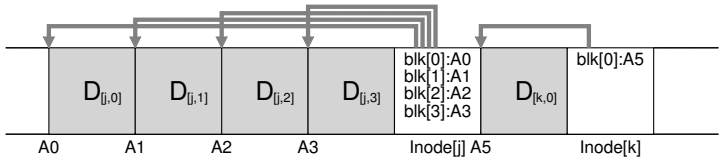
To achieve this end, LFS uses an ancient technique known as **write buffering**[1]. Before writing to the disk, LFS keeps track of updates in memory; when it has received a sufficient number of updates, it writes them to disk all at once, thus ensuring efficient use of the disk.

The large chunk of updates LFS writes at one time is referred to by the name of a **segment**. Although this term is over-used in computer systems, here it just means a large-ish chunk which LFS uses to group writes. Thus, when writing to disk, LFS buffers updates in an in-memory

[1] Indeed, it is hard to find a good citation for this idea, since it was likely invented by many and very early on in the history of computing. For a study of the benefits of write buffering, see Solworth and Orji [SO90]; to learn about its potential harms, see Mogul [M94].

segment, and then writes the segment all at once to the disk. As long as the segment is large enough, these writes will be efficient.

Here is an example, in which LFS buffers two sets of updates into a small segment; actual segments are larger (a few MB). The first update is of four block writes to file j; the second is one block being added to file k. LFS then commits the entire segment of seven blocks to disk at once. The resulting on-disk layout of these blocks is as follows:



43.3 How Much To Buffer?

This raises the following question: how many updates should LFS buffer before writing to disk? The answer, of course, depends on the disk itself, specifically how high the positioning overhead is in comparison to the transfer rate; see the FFS chapter for a similar analysis.

For example, assume that positioning (i.e., rotation and seek overheads) before each write takes roughly $T_{position}$ seconds. Assume further that the disk transfer rate is R_{peak} MB/s. How much should LFS buffer before writing when running on such a disk?

The way to think about this is that every time you write, you pay a fixed overhead of the positioning cost. Thus, how much do you have to write in order to **amortize** that cost? The more you write, the better (obviously), and the closer you get to achieving peak bandwidth.

To obtain a concrete answer, let's assume we are writing out D MB. The time to write out this chunk of data (T_{write}) is the positioning time $T_{position}$ plus the time to transfer D ($\frac{D}{R_{peak}}$), or:

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \tag{43.1}$$

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

$$R_{effective} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}}. \tag{43.2}$$

What we're interested in is getting the effective rate ($R_{effective}$) close to the peak rate. Specifically, we want the effective rate to be some fraction F of the peak rate, where $0 < F < 1$ (a typical F might be 0.9, or 90% of the peak rate). In mathematical form, this means we want $R_{effective} = F \times R_{peak}$.

At this point, we can solve for D:

$$R_{effective} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak} \tag{43.3}$$

$$D = F \times R_{peak} \times (T_{position} + \frac{D}{R_{peak}}) \tag{43.4}$$

$$D = (F \times R_{peak} \times T_{position}) + (F \times R_{peak} \times \frac{D}{R_{peak}}) \tag{43.5}$$

$$D = \frac{F}{1 - F} \times R_{peak} \times T_{position} \tag{43.6}$$

Let's do an example, with a disk with a positioning time of 10 milliseconds and peak transfer rate of 100 MB/s; assume we want an effective bandwidth of 90% of peak ($F = 0.9$). In this case, $D = \frac{0.9}{0.1} \times 100\ MB/s \times 0.01\ seconds = 9\ MB$. Try some different values to see how much we need to buffer in order to approach peak bandwidth. How much is needed to reach 95% of peak? 99%?

43.4 Problem: Finding Inodes

To understand how we find an inode in LFS, let us briefly review 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 easy, because they are organized in an array and placed on disk at fixed locations.

For example, the old UNIX file system 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; array-based indexing, given an inode number, is fast and straightforward.

Finding an inode given an inode number in FFS is only slightly more complicated, because FFS splits up the inode table 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.

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

TIP: USE A LEVEL OF INDIRECTION

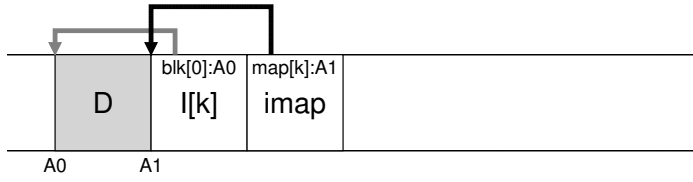
People often say that the solution to all problems in Computer Science is simply a **level of indirection**. This is clearly not true; it is just the solution to *most* problems (yes, this is still too strong of a comment, but you get the point). You certainly can think of every virtualization we have studied, e.g., virtual memory, or the notion of a file, 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. As Wheeler famously said, "All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections."

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.

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 reside on disk?

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

Instead, 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:



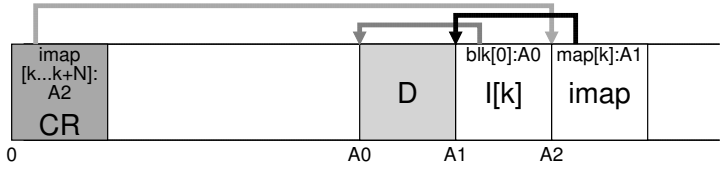
In this picture, the piece of the imap array stored in the block marked *imap* tells LFS that the inode k is at disk address $A1$; this inode, in turn, tells LFS that its data block D is at address $A0$.

43.6 Completing The Solution: The Checkpoint Region

The clever reader (that's you, right?) 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? In the end, there is no magic: the file system must have *some* fixed and known location on disk to begin a file lookup.

LFS has just such a fixed place on disk for this, known as the **checkpoint region (CR)**. The checkpoint region contains pointers to (i.e., addresses of) the latest pieces of the inode map, and thus the inode map pieces can be found by reading the CR first. 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 (which points to the latest pieces of the inode map); the inode map pieces each contain addresses of the inodes; the inodes point to files (and directories) just like typical UNIX file systems.

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. A real file system would of course have a much bigger CR (indeed, it would have two, as we'll come to understand later), many imap chunks, and of course many more inodes, data blocks, etc.



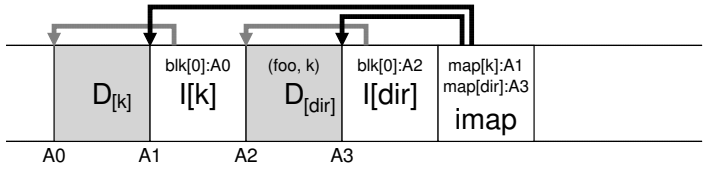
43.7 Reading A File From Disk: A Recap

To make sure you understand how LFS works, let us now walk through what must happen to 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. 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?

Thus far, we've simplified our discussion a bit by only considering inodes and data blocks. However, to access a file in a file system (such as `/home/remzi/foo`, one of our favorite fake file names), some directories must be accessed too. So how does LFS store directory data?

Fortunately, 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. Remember that LFS will do so sequentially on the disk (after buffering the updates for some time). Thus, creating a file `foo` in a directory would lead to the following new structures on disk:



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 accessing file `foo` (with inode number k), you would first look in the inode map (usually cached in memory) to find the location of the inode of directory dir ($A3$); you then read the directory inode, which gives you the location of the directory data ($A2$); reading this data block gives you the name-to-inode-number mapping of (`foo`, k). You then consult the inode map again to find the location of inode number k ($A1$), and finally read the desired data block at address $A0$.

There is one other serious problem in LFS that the inode map solves, known as the **recursive update problem** [Z+12]. 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.

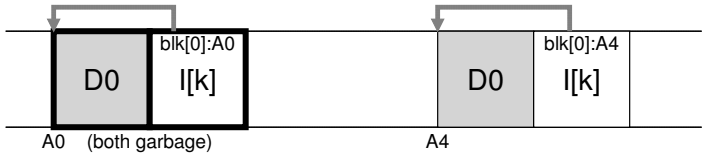
Specifically, whenever an inode is updated, its location on disk changes. If we hadn't been careful, this would have also entailed an update to the directory that points to this file, which then would have mandated a change to the parent of that directory, and so on, all the way up the file system tree.

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

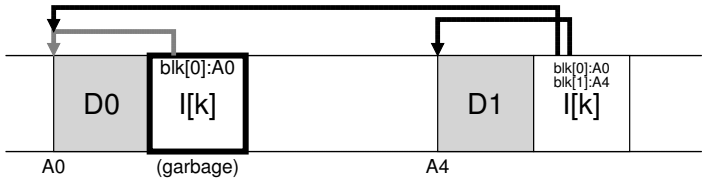
You may have noticed another problem with LFS; it repeatedly writes the latest version of a file (including its inode and data) to new locations on disk. This process, while keeping writes efficient, implies that LFS leaves old versions of file structures scattered throughout the disk. We (rather unceremoniously) call these old versions **garbage**.

For example, let'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 (note we omit the imap and other structures for simplicity; a new chunk of imap would also have to be written to disk to point to the new inode):



In the diagram, you can see that both the inode and data block have two versions on disk, one old (the one on the left) and one current and thus **live** (the one on the right). By the simple act of (logically) updating a data block, a number of new structures must be persisted by LFS, thus leaving old versions of said blocks on the disk.

As another example, 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:



So what should we do with these older versions of inodes, data blocks, and so forth? 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, and other structures, and **clean** them; cleaning should

thus make blocks on disk free again for use in subsequent writes. Note that the process of cleaning is a form of **garbage collection**, a technique that arises in programming languages that automatically free unused memory for programs.

Earlier we discussed segments as important as they are the mechanism that enables large writes to disk in LFS. As it turns out, they are also quite integral to effective cleaning. Imagine what would happen if the LFS cleaner simply went through and freed single data blocks, inodes, etc., during cleaning. The result: a file system with some 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.

Instead, the LFS cleaner works on a segment-by-segment basis, thus clearing up large chunks of space for subsequent writing. The basic cleaning process works as follows. 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.

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?

43.10 Determining Block Liveness

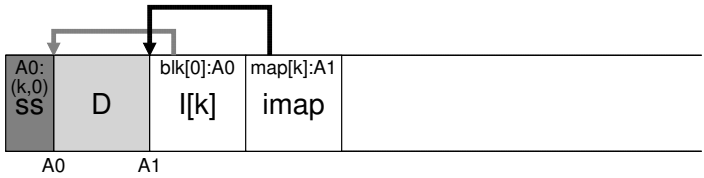
We address the mechanism first. Given a data block D within an on-disk segment S, LFS must be able to determine whether D is live. To do so, LFS adds a little extra information to each segment that describes each block. Specifically, LFS includes, for each data block D, its inode number (which file it belongs to) and its offset (which block of the file this is). This information is recorded in a 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 (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 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. A pseudocode summary of this

process is shown here:

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

Here is a diagram depicting the mechanism, in which the segment summary block (marked SS) records that the data block at address $A0$ is actually a part of file k at offset 0. By checking the imap for k, you can find the inode, and see that it does indeed point to that location.



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?

On top of the mechanism described above, LFS must include 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* segments. 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. 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 policy isn't perfect; later approaches show how to do better [MR+97].

43.12 Crash Recovery And The Log

One final problem: what happens if the system crashes while LFS is writing to disk? As you may recall in the previous chapter about journaling, crashes during updates are tricky for file systems, and thus something LFS must consider as well.

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. LFS also periodically updates the checkpoint region. Crashes could clearly happen during either of these operations (write to a segment, write to the CR). So how does LFS handle crashes during writes to these structures?

Let's cover the second case first. 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.

Let's now address the first case. 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. See Rosenblum's award-winning dissertation for details [R92].

43.13 Summary

LFS introduces a new approach to updating the disk. Instead of overwriting 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.

TIP: TURN FLAWS INTO VIRTUES

Whenever your system has a fundamental flaw, see if you can turn it around into a feature or something useful. NetApp's WAFL does this with old file contents; by making old versions available, WAFL no longer has to worry about cleaning quite so often (though it does delete old versions, eventually, in the background), and thus provides a cool feature and removes much of the LFS cleaning problem all in one wonderful twist. Are there other examples of this in systems? Undoubtedly, but you'll have to think of them yourself, because this chapter is over with a capital "O". Over. Done. Kaput. We're out. Peace!

The large writes that LFS generates are excellent for performance on many different devices. On hard drives, large writes ensure that positioning time is minimized; on parity-based RAIDs, such as RAID-4 and RAID-5, they avoid the small-write problem entirely. Recent research has even shown that large I/Os are required for high performance on Flash-based SSDs [H+17]; thus, perhaps surprisingly, LFS-style file systems may be an excellent choice even for these new mediums.

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** [HLM94], Sun's **ZFS** [B07], and Linux **btrfs** [R+13], and even modern **flash-based SSDs** [AD14], 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. In particular, WAFL got around cleaning problems by turning them into a feature; by providing old versions of the file system via **snapshots**, users could access old files whenever they deleted current ones accidentally.

References

[AD14] "Operating Systems: Three Easy Pieces" (Chapter: Flash-based Solid State Drives) by Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. Arpaci-Dusseau Books, 2014. *A bit gauche to refer you to another chapter in this very book, but who are we to judge?*

[B07] "ZFS: The Last Word in File Systems" by Jeff Bonwick and Bill Moore. Copy Available: `http://www.ostep.org/Citations/zfs_last.pdf`. *Slides on ZFS; unfortunately, there is no great ZFS paper (yet). Maybe you will write one, so we can cite it here?*

[H+17] "The Unwritten Contract of of Solid State Drives" by Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. EuroSys '17, April 2017. *Which unwritten rules one must follow to extract high performance from an SSD? Interestingly, both request scale (large or parallel requests) and locality still matter, even on SSDs. The more things change ...*

[HLM94] "File System Design for an NFS File Server Appliance" by Dave Hitz, James Lau, Michael Malcolm. USENIX Spring '94. *WAFL takes many ideas from LFS and RAID and puts it into a high-speed NFS appliance for the multi-billion dollar storage company NetApp.*

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

[MJLF84] "A Fast File System for UNIX" by Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry. ACM TOCS, Volume 2:3, August 1984. *The original FFS paper; see the chapter on FFS for more details.*

[MR+97] "Improving the Performance of Log-structured File Systems with Adaptive Methods" by Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, Thomas E. Anderson. SOSP 1997, pages 238-251, October, Saint Malo, France. *A more recent paper detailing better policies for cleaning in LFS.*

[M94] "A Better Update Policy" by Jeffrey C. Mogul. USENIX ATC '94, June 1994. *In this paper, Mogul finds that read workloads can be harmed by buffering writes for too long and then sending them to the disk in a big burst. Thus, he recommends sending writes more frequently and in smaller batches.*

[P98] "Hardware Technology Trends and Database Opportunities" by David A. Patterson. ACM SIGMOD '98 Keynote, 1998. Available online here: `http://www.cs.berkeley.edu/~pattrsn/talks/keynote.html`. *A great set of slides on technology trends in computer systems. Hopefully, Patterson will create another of these sometime soon.*

[R+13] "BTRFS: The Linux B-Tree Filesystem" by Ohad Rodeh, Josef Bacik, Chris Mason. ACM Transactions on Storage, Volume 9 Issue 3, August 2013. *Finally, a good paper on BTRFS, a modern take on copy-on-write file systems.*

[RO91] "Design and Implementation of the Log-structured File System" by Mendel Rosenblum and John Ousterhout. SOSP '91, Pacific Grove, CA, October 1991. *The original SOSP paper about LFS, which has been cited by hundreds of other papers and inspired many real systems.*

[R92] "Design and Implementation of the Log-structured File System" by Mendel Rosenblum. http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-696.pdf. *The award-winning dissertation about LFS, with many of the details missing from the paper.*

[SS+95] "File system logging versus clustering: a performance comparison" by Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan. USENIX 1995 Technical Conference, New Orleans, Louisiana, 1995. *A paper that showed the LFS performance sometimes has problems, particularly for workloads with many calls to `fsync()` (such as database workloads). The paper was controversial at the time.*

[SO90] "Write-Only Disk Caches" by Jon A. Solworth, Cyril U. Orji. SIGMOD '90, Atlantic City, New Jersey, May 1990. *An early study of write buffering and its benefits. However, buffering for too long can be harmful: see Mogul [M94] for details.*

[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes" by Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '13, San Jose, California, February 2013. *Our paper on a new way to build flash-based storage devices, to avoid redundant mappings in the file system and FTL. The idea is for the device to pick the physical location of a write, and return the address to the file system, which stores the mapping.*

Homework (Simulation)

This section introduces `lfs.py`, a simple LFS simulator you can use to understand better how an LFS-based file system works. Read the README for details on how to run the simulator.

Questions

1. Run `./lfs.py -n 3`, perhaps varying the seed (`-s`). Can you figure out which commands were run to generate the final file system contents? Can you tell which order those commands were issued? Finally, can you determine the liveness of each block in the final file system state? Use `-o` to show which commands were run, and `-c` to show the liveness of the final file system state. How much harder does the task become for you as you increase the number of commands issued (i.e., change `-n 3` to `-n 5`)?
2. If you find the above painful, you can help yourself a little bit by showing the set of updates caused by each specific command. To do so, run `./lfs.py -n 3 -i`. Now see if it is easier to understand what each command must have been. Change the random seed to get different commands to interpret (e.g., `-s 1`, `-s 2`, `-s 3`, etc.).
3. To further test your ability to figure out what updates are made to disk by each command, run the following: `./lfs.py -o -F -s 100` (and perhaps a few other random seeds). This just shows a set of commands and does NOT show you the final state of the file system. Can you reason about what the final state of the file system must be?
4. Now see if you can determine which files and directories are live after a number of file and directory operations. Run `tt ./lfs.py -n 20 -s 1` and then examine the final file system state. Can you figure out which pathnames are valid? Run `tt ./lfs.py -n 20 -s 1 -c -v` to see the results. Run with `-o` to see if your answers match up given the series of random commands. Use different random seeds to get more problems.
5. Now let's issue some specific commands. First, let's create a file and write to it repeatedly. To do so, use the `-L` flag, which lets you specify specific commands to execute. Let's create the file "/foo" and write to it four times: `-L c,/foo:w,/foo,0,1:w,/foo,1,1:w,/foo,2,1:w,/foo,3,1 -o`. See if you can determine the liveness of the final file system state; use `-c` to check your answers.
6. Now, let's do the same thing, but with a single write operation instead of four. Run `./lfs.py -o -L c,/foo:w,/foo,0,4` to create file "/foo" and write 4 blocks with a single write operation. Compute the liveness again, and check if you are right with `-c`. What is the main difference between writing a file all at once (as we do here) versus doing it one block at a time (as above)? What does this tell you about the importance of buffering updates in main memory as the real LFS does?
7. Let's do another specific example. First, run the following: `./lfs.py -L c,/foo:w,/foo,0,1`. What does this set of commands do? Now, run `./lfs.py -L c,/foo:w,/foo,7,1`. What does this set of commands do? How are the two different? What can you tell about the `size` field in the inode from these two sets of commands?

8. Now let's look explicitly at file creation versus directory creation. Run simulations `./lfs.py -L c,/foo` and `./lfs.py -L d,/foo` to create a file and then a directory. What is similar about these runs, and what is different?
9. The LFS simulator supports hard links as well. Run the following to study how they work:
   `./lfs.py -L c,/foo:l,/foo,/bar:l,/foo,/goo -o -i`.
   What blocks are written out when a hard link is created? How is this similar to just creating a new file, and how is it different? How does the reference count field change as links are created?
10. LFS makes many different policy decisions. We do not explore many of them here – perhaps something left for the future – but here is a simple one we do explore: the choice of inode number. First, run `./lfs.py -p c100 -n 10 -o -a s` to show the usual behavior with the "sequential" allocation policy, which tries to use free inode numbers nearest to zero. Then, change to a "random" policy by running `./lfs.py -p c100 -n 10 -o -a r` (the `-p c100` flag ensures 100 percent of the random operations are file creations). What on-disk differences does a random policy versus a sequential policy result in? What does this say about the importance of choosing inode numbers in a real LFS?
11. One last thing we've been assuming is that the LFS simulator always updates the checkpoint region after each update. In the real LFS, that isn't the case: it is updated periodically to avoid long seeks. Run `./lfs.py -N -i -o -s 1000` to see some operations and the intermediate and final states of the file system when the checkpoint region isn't forced to disk. What would happen if the checkpoint region is never updated? What if it is updated periodically? Could you figure out how to recover the file system to the latest state by rolling forward in the log?