Log-Based File Systems

1 Introduction

Hello, and welcome to this lesson on log-based file systems.

In the more recent past memory has become plentiful and cheap, and operating system designers have been thinking about how to best leverage this to improve file system performance. One result of this was the development of log-structured file systems, which we will explore in this lesson.

We will first start with an observation, made in the early Nineties already, that memory has been getting cheaper, and systems are equipped with significant amounts of memory, which allows to shield the overall system from the performance limitations of slow storage devices through caching and write buffering.

In the meantime, the UNIX Fast File System, which was designed with the optimized use of slow hard disk drives in mind, was not able to leverage large caches.

New file system organizations were explored, most prominently the Log-structured File System (LSF), by Rosenblum and Ousterhout at the University of California at Berkeley.

We will look at the principles underlying the operation of LFS, namely write-buffering and sequential writing.

While the principles appear simple once they are described, the devil is in the detail. We will look at a number of issues that need to be addressed by the designer of a log-structured file system.

At the end of this lesson you will understand the principles underlying log-structured file systems and a few of the practical issues that need to be addressed.

Let's begin.

It became apparent in the early Nineties that technologies over the last decades had developed very unevenly.

If we plot the performance of system components over the years, one can observe that

CPU Performance increased exponentially.

RAM size increased exponentially as well.

As always, disk performance has two aspects.

The transfer bandwidth increased significantly, with the advent of high performance busses and then even more so with RAID.

The disk access latency, on the other hand, did not really improve much over the years.

2 More RAM leads to ...

The massive increase of RAM in systems had a significant effect on how disks were used.

In particular because they allowed for large file caches.

Instead of having all disk READ and WRITE operations go to disk,

The available memory now enables the designer to use large portions of it as file cache.

In this fashion the cache can handle much of the READ requests.

Since WRITES sill go to disk, WRITES dominate the disk traffic. File system optimizations therefore have to target the handling of disk WRITE requests.

The WRITE performance can be improved by using some of the cache memory as write buffer.

The write buffer would amortize the overhead of writing individual blocks disk by buffering a large number of WRITES before writing them to disk.

This would allow, in principle, to write a large sequence of blocks to disk sequentially rather than randomly. This make the overall write operations much more efficient.

Of course, this performance comes at a cost, as the write cache memory is volatile.

If there are blocks in the write buffer, and the system crashes, the blocks are lost.

3 Problems with Berkeley UNIX FFS

In comparison, one of the most advanced file systems at the time could not at all take advantage of file caches and write buffers, for several reasons.

The first problem was that, while it tried to lay out the file data somewhat sequentially, with the use of cylinder groups, but blocks of files are physically separated. They may be in the same cylinder group, but scattered within it.

Similarly, the inodes and data blocks of the same file are separated as well, because FFS has an inode list, which is separate from the data block area of the cylinder block.

Finally, directories are stored in separate files from the data files. This separation makes file operations very expensive.

Let's look at an example where a new file is created. Here we have a simplified picture of the cylinder group, with a single-block directory highlighted.

1. First we need to create an inode for the new file. We call it I_F.
2. Then we mark the inode in the inode bitmap block.
3. Third, we find a data block and write the data, marked as D_F here.
4. We update the data block bit map.
5. After that we update the directory file.
6. Finally, we need to update the inode of the directory file.

In summary, the creation of a single file causes 6 WRITE operations.

Experiments showed that for writes to small files, less than 5% of disk bandwidth was used for user data.

To make matters worse, many of FFS's WRITE operations are synchronized, meaning that these are blocking reads, and therefore cannot be really leverage write buffers.

While file data is written synchronously, and therefore can be pushed to write caches in principle,

Meta data, including inodes, directory files, and bitmap blocks, has to be written to the disk directly.

4 Log-Structured File Systems

Log-structured file systems are structured such that they take full advantage of caching and write buffering.

The fundamental idea is to focus on WRITE performance, as READs are typically served from the cache.

All changes are buffered in the cache.

And the buffered WRITE operations are sent disk sequentially. As a result, the many little random WRITEs that one would encounter in a system like FFS are aggregated into large sequential writes that occur asynchronously.

4.1 How to Write Sequentially

Let's illustrate how write buffering and sequential writes work.

We represent the disk as a stream of blocks that we can write to.

The gray portion is used already.

Writes are done to a cache with a write buffer.

If we create a new file, we write the data block of the new file to the write buffer.

Next we write the new file inode to the buffer. We note that we are not caching the entire inode block, but rather just the inode.

Next comes the updated directory data block, and last the updated directory inode.

At some point, the data in the buffer is pushed to disk in one swoop, sequentially, that is, in a sequence of blocks on disk.

If sometimes later the file is modified, the updated data bock and inode are written to the buffer, and at some time later pushed to disk, just after the previous blocks.

If we create another file, say G, in the same directory, we write the data block and the new inode to the write buffer, and the modified directory data and the updated directory inode as well. As before, after some time, the WRITEs are pushed to disk sequentially.

As the data is pushed to disk, the file system maintains a pointer to the end of the last WRITE. A0 in this case.

The allocation data in the inode points to the location of the data block, which is A0.

As we write, the end pointer is updated, to A1 in this case, the location of the updated block in the directory file, say block X, is therefore A1. and so on.

4.2 Issues

Now we know the basic mechanism behind log-structured file systems. But there are several important issues to address to make this all work.

First, how do we even read from the log? In order to read the data in a file we need to find and read the inode, to get to the allocation information. Once the inode is found, and the allocation information available, the rest of the read operations are quite identical to those of other file systems such as the UNIX Fast File System.

The problem is that inodes are scattered all over the disk, and typically there are multiple copies of the inode for a specific file.

The solution is to use so-called "inode maps", which are special blocks. that link together the inodes. A READ operation would therefore use the inode map to locate the inode and then go from there to read the file data.

In Other Words: File Location and Reading Let's put this differently: In order to read data from a log file, typically one has to scan it.

LFS adds an index structure in form of inode maps to significantly speed up the scanning process, basically turning it into a random access of the desired inode.

The structure of the inodes is identical to those of other file systems, such as FFS, which means that, once the inode is found, the rest of the read operations to the file are the same as for other file systems.

Differently from other file systems, however, is that the inode location is not fixed. As we saw in the previous example, new versions of the same inode are stored in different locations on disk. The inode map ties provides access to the inodes. The inode map is largely stored in memory, in order to speed up the mapping of files to their inodes.

4.3 How to Write Sequentially: Issues II

Now that we have a better understanding of how to read data from the log.

The second issue is how to WRITE the data to the log. Specifically, how does one identify a sequence of free blocks that is long enough to hold the blocks for the next push.

Let's assume that in the last push of blocks to disk we reached the end of the disk. and we now need to push another sequence of blocks.

The new sequence clearly does not fit in the remaining space.

Let's notice, however, that there is much data on the disk that is not current anymore. For example, the original file data block has been overwritten, as has the directory block, including both inodes.

These items are not current anymore, and can therefore be deleted. and the space reclaimed for new blocks.

5 Free-Space Management

The question is how to maintain segments of free space that are sufficiently long to fit the sequential writes of logs?

One solution is to identify holes and to thread, so-to-say, the new log through these holes. In addition to being complicated and to scatter writes are over the disk, this solution also has to deal with fragmentation issues.

Alternatively one could de-fragment the disk space and so compact the live data. The problem with this approach is the cost of copying the live data around the disk.

The solution selected by SFS is to use fixed-sized holes, which are called "segments". Whenever the system is low on free segments, new segments are created by cleaning existing segments.

6 Garbage Collection: Segment Cleaning

This is also called garbage collection. Let's see how this is done.

Say the current portion of the disk fits in a segment.

The four items on the right are not current and can be deleted. The rest of the data is live.

We clean this segment by first identifying a free segment and then by copying the live data over from the old one.

As we do that we update the inode information to reflect the fact that the data blocks have been copied to a new location. The inode map of course is updated as well.

The stale data can be deleted and the old segment now becomes free. By copying only the live data of multiple segments to fewer free segments we increase the total number of free segments.

In summary, we compact the live data by, reading the content of a number of segments into memory, then identifying the live data in these segments, and by finally writing the live data back to a smaller number of segments.

How do we even know what data is live? We keep for each segment a so-called segment summary block, which keeps track of what data is live and which one can be reclaimed.

Finally, you may have noticed that we never mentioned free-block lists or bitmaps. This is because we don?t need them. New space for blocks is generated through this garbage collection process.

7 Log-Structured File Systems: Summary

With this we have come to the conclusion of our exploration of Log-structured file systems.

The development of this form of file systems was triggered by the availability of large amounts of memory that could be used to cost-effectively do file caching.

Traditional file systems could not take full advantage of caches.

The idea of log-structured file systems is to focus on WRITE performance.

WRITE operations are cached in memory and pushed to disk sequentially.

The principle behind log-structured file systems is relatively simple, but a number of tricky issues need to be addressed to make it work. Among them are the identification of live inodes to support reads, the management of free block sequences to support sequential writes. For the latter, LFS uses a garbage collection approach that we encounter again in a similar fashion in the management of NAND flash memory used in Solid State Disks. We have not touched a number of implementation issues, such as how to keep track of inode maps and others. Many of these are pretty standard data structures issues, and solutions become readily apparent once one starts reasoning about an implementation approach.

We sincerely hope that you enjoyed this lesson on Log-structured file systems.

While log-structured file systems have not become mainstream for general purpose file systems, they have had an influence on the design of file systems for NAND Flash memory used in Solid State drives and for optical rewriteable storage devices.