Log-Structured File Systems

Annalise Tarhan

March 17, 2021

1 Run ./lfs.py -n 3. Can you figure out which commands were run to generate the final file system contents? Can you tell which order those commands were issued in? Finally, can you determine the liveness of each block in the final file system state? How much harder does the task become as you increase the number of commands issued?

Seed 1
create file /tg4
write file /tg4 offset=6 size=0 (the offset number seems nonsensical since no blocks were written)
create file /lt0

Seed 2
create file /vt6
write file /vt6 offset=4 size=4
create file /ks9

Seed 3
create file /jp6
create file /vg2
link file /jp6 /mq1

Seed 4
create file /ke0
write file /ke0 offset=6 size=6 (I think this is broken. Supposedly six blocks were written, but there was only room for two.)
create file /he1

Except for the bizarre offset/size behavior, the log clearly reflects the commands. It isn't difficult to determine the commands, the order they were given in, or which blocks are live. Increasing the number of commands doesn't make the task disproportionately harder.

2 Repeat the previous problem with the -i flag.

Seed 5
create directory /uy7
create file /yq9
create file /go5
create file /hx7
create file /uy7/dq1

Seed 6
create file /rm7
write file /rm7 offset=2 size=6
delete file /rm7
create directory /ou2
create directory /ou2/ic8

Seed 7
create file /bn3
create file /al0
create file /vd2
create directory /pk9
create file /pk9/hd1

3 Run with the -F flag, which doesn't show the final state of the file system.

Seed 100
0 - live - checkpoint: 10
1 - [.,0] [..,0]
2 - type:dir size:1 refs:2 ptrs:1
3 - chunk(imap): 2

4 - live - [.,0] [..,0] [us7,1]
5 - live - type:dir size:1 refs:2 ptrs:4
6 - type:reg size:0 refs:1 ptrs:
7 - chunk(imap): 5 6

(skipped because a write of size 0 doesn't make sense)
8 - live - xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy

```
9 - live - type:reg size:1 refs:1 ptrs: – – – – – – – 8
10 - live - chunk(imap): 5 9
```

4 Now see if you can determine which files and directories are live after a number of file and directory operations. Run with the following flags and then examine the final file system state. Can you figure out which pathnames are valid?

Seed 8
/nn0
/ag4
/gz0
/ri6/wh3 /ri6/yv5/ha7 ri6/er9
/ym1

Seed 9
/fh8
/wb3
/dp8
/co0
/xk3
/pe6/pd2
/sy2
/jr5/

5 First, create a file and write to it repeatedly. See if you can determine liveness of the final file system state.

```
0 - live - checkpoint: 19
4 - live - [.,0] [..,0] [foo,1]
5 - live - type:dir size:1 refs:2 ptrs: 4
8 - live - v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0
11 - live - t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0
14 - live - k0k0k0k0k0k0k0k0k0k0k0k0k0k0k0k0
17 - live - g0g0g0g0g0g0g0g0g0g0g0g0g0g0g0g0
18 - live - type:reg size:4 refs:1 ptrs: 9 11 14 17
19 - live - chunk(imap): 5 18
```

6 Now do the same thing but with a single write operation instead of four. Compute the liveness again. What is the main difference between writing a file all at once versus doing it one block at a time? What does this tell you about the importance of buffering updates in main memory as the real LFS does?

```
0 - live - checkpoint: 13
4 - live - [.,0] [..,0] [foo,1]
5 - live - type:dir size:1 refs:2 ptrs: 4
8 - live - v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0
9 - live - t1t1t1t1t1t1t1t1t1t1t1t1t1t1t1t1
10 - live - k2k2k2k2k2k2k2k2k2k2k2k2k2k2k2k2
11 - live - g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3
12 - live - type:reg size:4 refs:1 ptrs: 8 9 10 11
13 - live - chunk(imap): 5 12
```

There are the same number of live blocks, but there are many more dead ones in between. The disk space for those dead blocks is wasted, so reads and clean up will both take longer. Buffering updates in memory and writing all at once avoids that waste.

7 Run with the following two sets of flags. What does each set of commands do? How are they different? What can you tell about the size field in the inode from these two sets of commands?

The first creates a file called foo, then ‘writes’ 0 blocks to it. The second creates a similar file, then writes a block at offset seven. Unsurprisingly, the final size of the first file is zero, but the final size of the second file is eight, even though only one block was written. This implies that the size of the file includes the empty space before the block that is actually written to.

8 Now let's look explicitly at file creation versus directory creation. Run the following simulations. What is similar about these runs and what is different?

Both simulations create a file and write a block to it; the difference is where the block is written. In the first simulation, it is written at the beginning, offset zero. In the second, it is written at the end, offset seven. The size of the first file is one block, while the size of the second is eight blocks.

9 Run the following to study how hard links work. 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?

When a hard link is created, a new entry is added to the parent directory, just like with a newly created file, but instead of creating a new inode, the new file points to the original file's inode and the inode's reference count is incremented.

10 Run the following 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. 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?

The most obvious difference is that the sequential version writes to 43 blocks while the random version writes to 53, since sparsely populated imaps are spread across multiple blocks, wasting a significant amount of space. Clearly, to avoid wasting space, inode numbers should be chosen to be as densely packed as possible.

11 In the real LFS, the checkpoint region is only updated periodically to avoid long seeks. Run the following command 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?

If the checkpoint region is never updated, the entire log would need to be read to find the most current data. Periodic updates guarantee that at most one period's worth of log updates need to be read. Rolling forward in the log will allow you to recover the file system's latest state if all updates were successful.