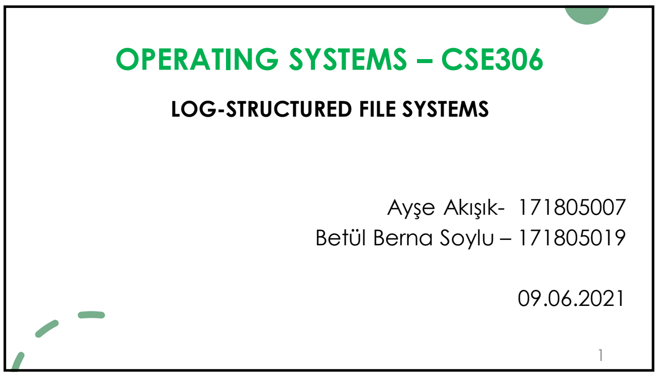


OPERATING SYSTEMS – CSE306

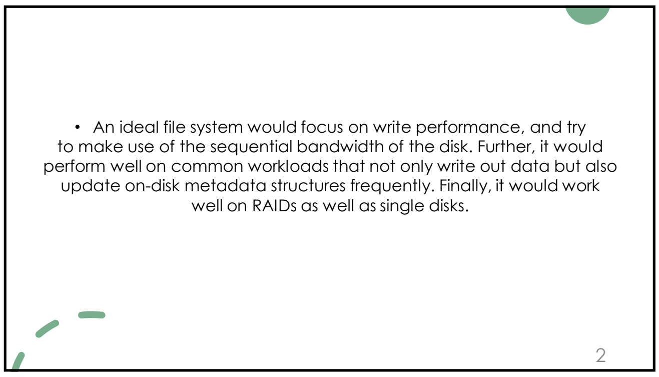
LOG-STRUCTURED FILE SYSTEMS

Ayşe Akışık- 171805007
Betül Berna Soylu – 171805019

09.06.2021

1

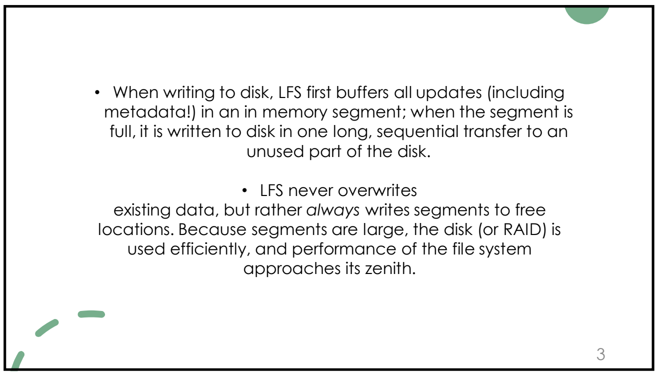
1



- An ideal file system would 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.

2

2

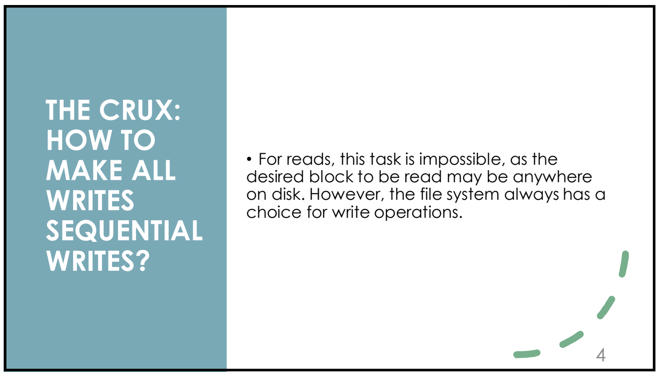


- When writing to disk, LFS first buffers all updates (including metadata!) in an in memory segment; when the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk.

- 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.

3

3

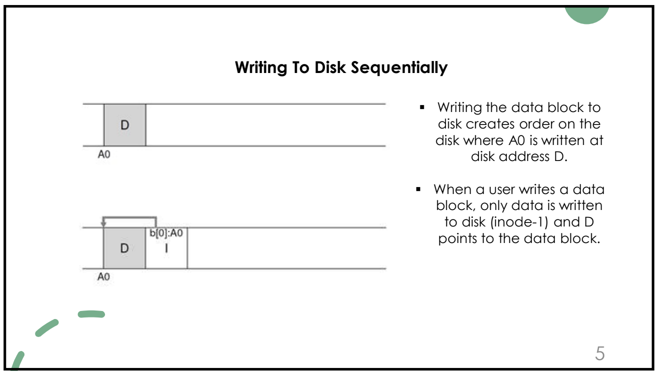


THE CRUX: HOW TO MAKE ALL WRITES SEQUENTIAL WRITES?

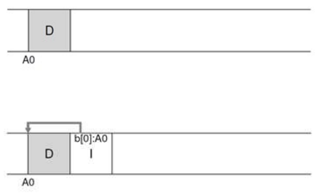
- For reads, this task is impossible, as the desired block to be read may be anywhere on disk. However, the file system always has a choice for write operations.

4

4



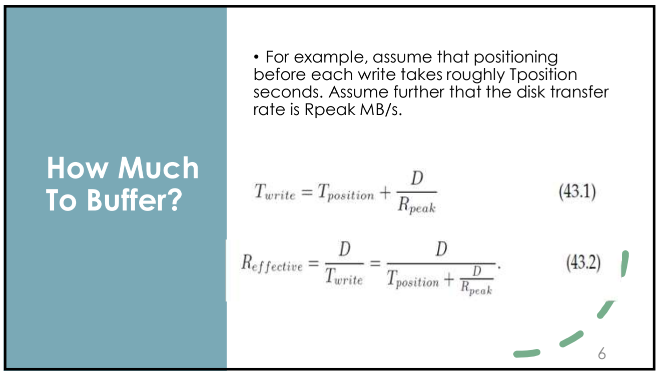
Writing To Disk Sequentially



- Writing the data block to disk creates order on the disk where A0 is written at disk address D.
- When a user writes a data block, only data is written to disk (inode-1) and D points to the data block.

5

5



How Much To Buffer?

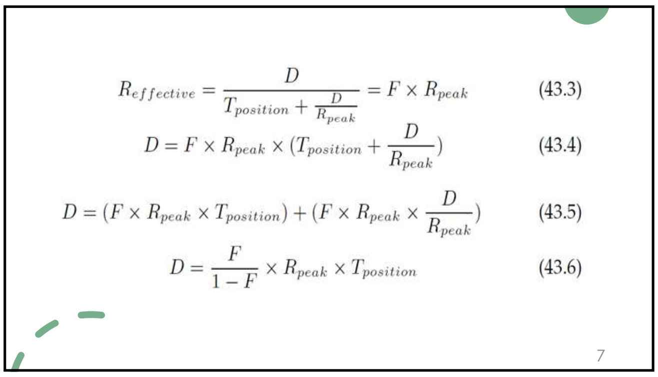
- For example, assume that positioning before each write takes roughly Tposition seconds. Assume further that the disk transfer rate is Rpeak MB/s.

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \qquad (43.1)$$

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

6

6



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

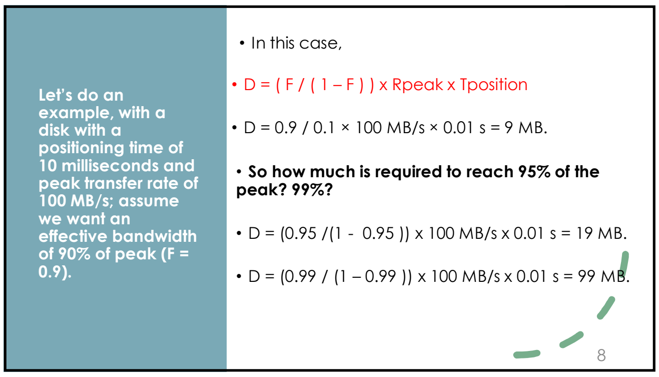
$$D = F \times R_{peak} \times (T_{position} + \frac{D}{R_{peak}}) \quad (43.4)$$

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

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

7

7

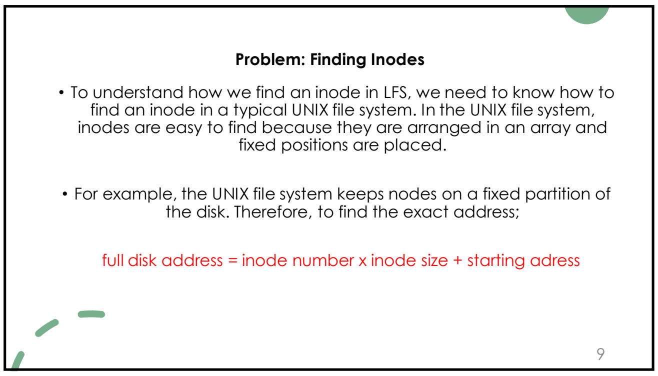


Let's do an example, with a disk with a positioning time of 10 milliseconds and peak transfer rate of 100 MB/s; assume we want an effective bandwidth of 90% of peak (F = 0.9).

- In this case,
- D = (F / (1 – F)) x Rpeak x Tposition
- D = 0.9 / 0.1 × 100 MB/s × 0.01 s = 9 MB.
- **So how much is required to reach 95% of the peak? 99%?**
- D = (0.95 /(1 - 0.95)) x 100 MB/s x 0.01 s = 19 MB.
- D = (0.99 / (1 – 0.99)) x 100 MB/s x 0.01 s = 99 MB.

8

8



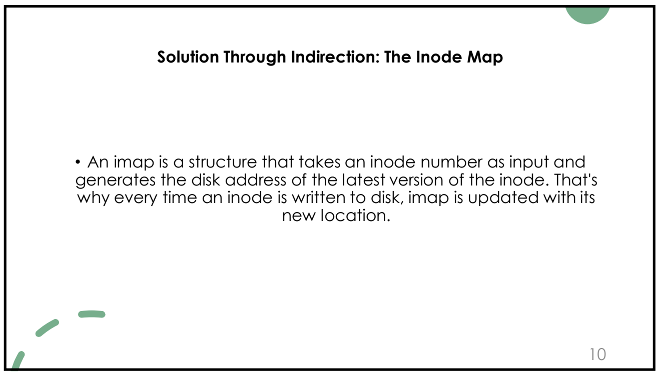
Problem: Finding Inodes

- To understand how we find an inode in LFS, we need to know how to find an inode in a typical UNIX file system. In the UNIX file system, inodes are easy to find because they are arranged in an array and fixed positions are placed.
- For example, the UNIX file system keeps nodes on a fixed partition of the disk. Therefore, to find the exact address;

full disk address = inode number x inode size + starting adress

9

9

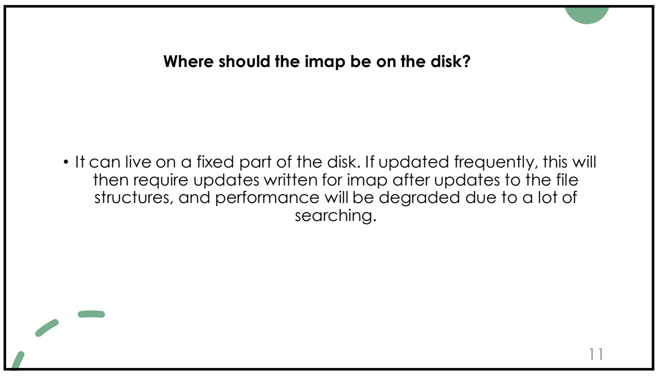


Solution Through Indirection: The Inode Map

- An imap is a structure that takes an inode number as input and generates the disk address of the latest version of the inode. That's why every time an inode is written to disk, imap is updated with its new location.

10

10

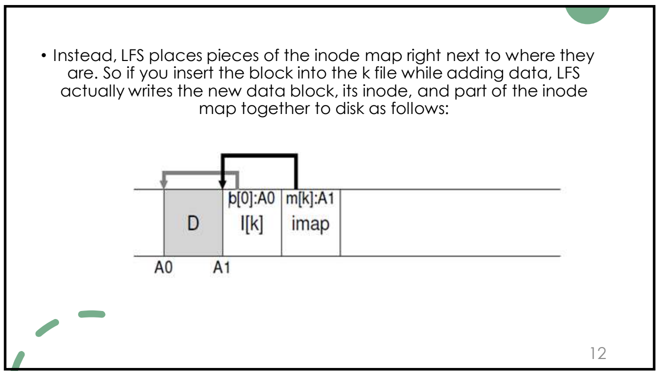


Where should the imap be on the disk?

- It can live on a fixed part of the disk. If updated frequently, this will then require updates written for imap after updates to the file structures, and performance will be degraded due to a lot of searching.

11

11



- Instead, LFS places pieces of the inode map right next to where they are. So if you insert the block into the k file while adding data, LFS actually writes the new data block, its inode, and part of the inode map together to disk as follows:

D
b[0]:A0
I[k]
m[k]:A1
imap
A0
A1

12

12

Completing The Solution: The Checkpoint Region

So how do we find the inode map, even though its parts are also spread to the disk?

- LFS has a fixed location on the disk known as the checkpoint zone (CR) for this.

- The control point region contains pointers (ie addresses) to the last parts of the inode map so that the inode map parts can be found by first reading the CR.

13

13

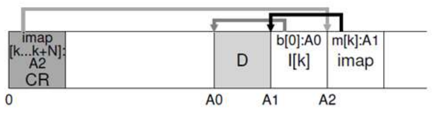
- The checkpoint zone is updated periodically and therefore affects performance badly.

- Each of the inode mapping pieces contains the addresses of the nodes.

14

14

- There is a sample of the checkpoint region (at the beginning of the disk, address 0) and a single imap part, inode and data block.



15

15

Reading A File From Disk: A Recap

1) To begin, suppose we have nothing in memory.
- The first data structure on the disk we need to read is the checkpoint region.

2) The checkpoint region contains pointers (i.e. disk addresses) to the entire inode map, and therefore LFS then reads the entire inode map and caches it in memory.

16

16

3) After that, when a file's inode number is given, LFS looks for the inode number to the **inode-disk-address** mapping in imap and reads the latest version of the inode.

4) To read a block from the file, at this point, LFS proceeds using direct pointers or indirect pointers as needed.

- In the general case, LFS should perform the same number of I / O when reading a file from disk; all imap is cached and so the job of LFS during a read is to look for the address of the inode in imap.

17

17

What About Directories?

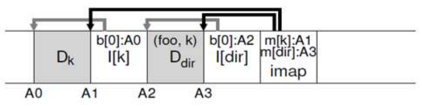
- Some directories also need to be accessed to access a file (such as home / remzi / foo) on a file system.

So how does LFS store directory data?

- For example, when creating a file on disk, LFS must write both a new inode, some data, directory data, and an inode referencing that file.
- LFS will do this sequentially on disk. Therefore, creating a 'foo' file in a directory results in the following new structures on the disk:

18

18



- Part of inode map contains information about the location of the site, both the index file and the newly created file 'f'. Therefore, when accessing the 'foo' file (with the inode number k), we first look at the inode map (usually cached in memory) to find the location of the inode (A3) of the directory.
- After that, we read the inode directory, which gives you the location of the index data (A2). Reading this data block gives (foo, k) a name-to-inode mapping. Then we refer to the inode map again to find the location of the inode number k (A1) and finally read the desired data block at address A0.

19

19

- The solution of LFS is;

Even if the position of an inode changes, the change should never be reflected in the array itself; instead, the imap structure is updated while the directory keeps the same **name-inode-number** mapping. Indirectly, LFS eliminates the recursive update problem.

20

20

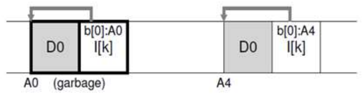
A New Problem: Garbage Collection

- LFS transfers the latest version of a file (including inode and data) to new locations on the disk. This process means that it leaves older versions scattered across the disk. We also call old version data garbage.

21

21

- For example, we have a file referenced by the inode number k that points to a single data block D0.
- By updating this block, we are creating both a new inode and a new data block.



- In the diagram, we can see both the inode and the data block on disk, one old (left) and one current and therefore live (right) versions.

22

22

- **Well, these are old inode versions, data blocks, etc. What should we do with it?**

- These old versions can be kept around and allow users to restore old file versions; Such a file system is known as a "versioning file system" because it keeps track of different versions of a file.
- However, LFS instead stores only the latest live version of a file; Therefore, in the background, LFS must periodically find and clean these old dead versions of file data, nodes, and other structures.

23

23

Determining Block Liveness

- Given a data block D within disk segment S, LFS must be able to determine if D is alive. To do this, LFS adds a little extra information to each partition that defines each block. Specifically, LFS contains, for each data block D, the inode number (the file to which it belongs) and its offset (which block of the file is).
- Specifically, LFS contains, for each data block D, the inode number (the file to which it belongs) and its offset (which block of the file is). This information is recorded in a structure at the head of the segment known as the segment summary block.

24

24

- With this information, it is easy to determine whether the block is alive or dead.

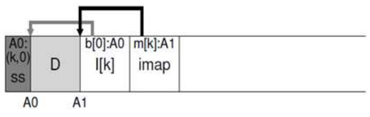
1) For a D block on disk at address A, look at the segment summary block and find the inode number N and offset T.
2) Next, look at imap to find where N lives and N should be read from disk.
3) Finally, using the offset T, the inode must be looked at to see where the Tth block of this file is on disk.

- If it points to the exact A disk address, LFS can conclude that the block D is live. If he points elsewhere, we can understand LFS, D is not alive.

pseudo code summary:

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

- The part mechanism of the segment summary block (SS), the data block at address A0,actually a part of the 'k' file at offset 0 is as follows:



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

- It is more difficult to determine which blocks to clean.
- In the original LFS article [RO91], the authors take an approach trying to separate hot and cold sections.
- The **hot section** is a section where content is often overwritten, so a long wait is required before cleaning.
- A **cold section** can have a few dead blocks, Therefore, authors should clear cold areas earlier and hot areas later.

Crash Recovery And The Log

- During normal operation, LFS writes buffers to a sector and writes the segment to disk when the segment is full or when a certain amount of time has elapsed.
- LFS organizes these transcriptions in a log, that is, the checkpoint region points to the head and tail segment, and each segment points to the next segment to be written. LFS also regularly updates the checkpoint zone. Crashes can obviously occur during any of these processes.

How do LFS crashes work at the time of writing?

- To make sure that the CR update happens atomically, LFS actually holds two CRs, one at each end of the disk, and writes them alternately.
- LFS also implements a protocol while updating the CR with the latest pointers to inodemap and other information; writes first a header (with timestamp), then the body of the CR, and finally a final block (also with a timestamp).
- If the system crashes during a CR update, LFS can detect this by seeing an inconsistent pair of time stamps. LFS always chooses to use the latest CR with consistent timestamps, thus ensuring that the CR is updated consistently.