Log Structured File System: Disk Organization

Charmi Shah
Department of Computer Science
Engineering
Oakland University
Rochester Hills, Michigan USA
charmishah@oakland.edu

Riya Singh
my Department of Computer Science
Engineering
Oakland University
Rochester Hills, Michigan USA
riyasingh@oakland.edu

Abstract

According to previous research, a log-structured file system (LFS) has the potential to significantly outperform typical UNIX FFS in terms of write performance, recovery time, and file creation and deletion rates.Will reading research paper we came across lots of theory of how file system work but it was really difficult to visualize how actually think happen in the backend of file system so, Through this paper, we are trying a simple implementation of a log-structured file system imitation with a focus on disk organization where the user will be able to get the text file from a Linux or mac system and will be able to read, update, and can delete the file from the log-structured file system. Here buffer cache will be handled by the host operating system memory management.

1. Introduction

A research group at UC Berkeley led by Professor John Ousterhout and graduate student Mendel Rosenblum developed a new file system known as the log-structured file system. Their motivation behind this new file system development was based on the following observations:[1]

1. System memories are growing rapidly
2. There is a large gap between random I/O performance and sequential I/O performance
3. Existing file systems perform poorly on many common workloads and
4. File systems are not RAID-aware.

Log-structured is a disk storage management system, which assumes that files are cached in the main memory and that increasing memory size will make the cache more effective with satisfying read requests. A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both the writing and crash recovery of the files. On disk, the log structure contains indexing information so the files can be read back from the log efficiently. In comparison to some other file system that uses a log-based system as a temporary storage option, log-structured file systems store data permanently in log format on the disk, which is structured for such usage.[1]

The log-structured file system uses the full potential of the disk array even if the file sizes are small, apart from the log-structured file system's excellent ability to use full disk bandwidth it has other benefits as well, like Fast Recovery: The sequential nature of the log-based system allows much faster crash recovery: current Unix file systems typically must scan the entire disk to restore consistency after a crash, but a log-structured file system need only examine the most recent portion of the log and Versioning: log-structured file systems keep the older version intact, so in case of if the older version needs to be made available to the user it could be achieved by saving it in slower mass storage section on disk.[2]

In recent years, CPU speed has improved by leaps and bounds, simultaneously the memory size also has increased but probably not as fast as the CPU speed. Contrary to this, disk speed has not improved at such a pace. This shift will open new CPU-intensive applications development but suggests that new I/O techniques will be needed to keep up the I/O from being a performance bottleneck.[5]

2. Problems and challenges

Modern file systems struggle to adapt to the technology and workloads of the 1990s due to two basic issues. First, they distribute data over the disk in a way that results in an excessive number of quick accesses. The Berkeley Unix fast file system (Unix FFS), for instance, performs an excellent job at ordering the files sequentially on the disk, but it physically splits the different files. Additionally, a file's characteristics ("inode") and directory entry carrying the file's name are distinct from the file's contents. In UNIX FFS, it takes five I/O's such as directory data, directory attributes, files data, and two file attributes for writing the new file, which in turn leads to only 5 percent of the time spent in the utilization of disk potential, while most of the effort is spent behind seeking.[1]

The second issue with modern file systems is that they frequently write synchronously, requiring an application to wait for the write to finish before proceeding rather than handling the write in the background. Whereas file system information structures like directories and inodes are written synchronously by UNIX FFS and file data blocks are written asynchronously by UNIX FFS. For the synchronous writes of small multiple files when coupled with disk performance it tends to make it harder for the application to take advantage of the CPU performance as they would be dominated by the synchronous writing of files at the same time. They also prevent the file cache from being used as a write buffer.[1]

To increase file, I/O performance, several systems have adopted file caching. The goal of file caching is to save recently used disk blocks in main memory so that further disk transfers for the same information can be avoided. File caching significantly lowers disk I/O performance because of the Locality in file access patterns. Nonetheless, file caching results in two main issues.

1. Future disk traffic will be increasingly dominated by writes, with most of the written data will remain in the file cache until it is overwritten or erased. Disk I/O is mostly performed as a safety measure in case the cache contents are lost.

2. Only 3 percentage of the available raw disk bandwidth may be used if files are only 3 to 4 bytes long and two transfers are needed for each file. The only way to increase disk bandwidth is by drastically lowering the number of seeks per file accompanied by reduced file size.[5]

3. Log-structured file system Model

An ideal file system prioritizes write performance and tries to use the sequential bandwidth of the disk. Additionally, it will function effectively under typical workloads that often alter the metadata structure on disk in addition to writing data.

Rosenblum and Ousterhout developed a brand-new form of file system called LFS or Log-Structured File System. When writing to the disk, LFS first buffers all changes which also includes metadata, and kept together into an in-memory segment. When the segment is full, it is written to the disk in a long pass that is sequential to an empty area of the disk. LFS always writes segments to vacant slots instead of overwriting existing data. The huge segments allow for effective utilization of the disk and optimal file system efficiency.[1]

3.1 DISK organization :

Here we are trying to divide the disk into several segments and superblocks each segment consists of a bunch of blocks and segment tables.

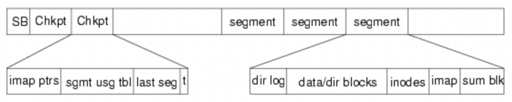


Figure 1. log-structured disk layout [6]

Above figure Fig 2 is a log-structured disk layout. log-structured allows a fast restart from the crash with

1. locating unused segments to write
2. last and next pointer in summary allows recovery
3. Imap pointer helps finding inodes
4. Time stamp allows locating last successful checkpoint.

3.1.1 SEGMENT

The first issue with a log-structured file system is how to convert all file system state modifications into a sequence of sequential disk writes. Assume we are writing a data block d to a file; let's say d is written at address A0. However, when a user writes a data block, additional information which is metadata that must be updated and changed. Additionally, we set aside some room for the inode table, which contains a variety of on-disk inodes. To point to the data block d, we write the file's inode to disk. The core of LFS is the simple concept of writing each update to the disk one at a time. As a result, we may issue many contiguous writes (or one massive write) to the drive-in to get high write performance. Simply writing to the disk in sequential order is not sufficient to reach peak performance. So, a big in-memory segment serves as a buffer for all writes (data and metadata) (typically of some MB). There are several data blocks and inodes inside a segment. It is written to the append-only log when it is full. Specifically, each section is made up of inodes, inode maps, and segment summary that contains the pointer to the next summary of the next segment and helps to identify for file block: the inode number and relative block number.[1]

The layout of the log-structured file system shows storage has been divided into segments regardless of the block type, for example, inodes, data blocks, and direct blocks.

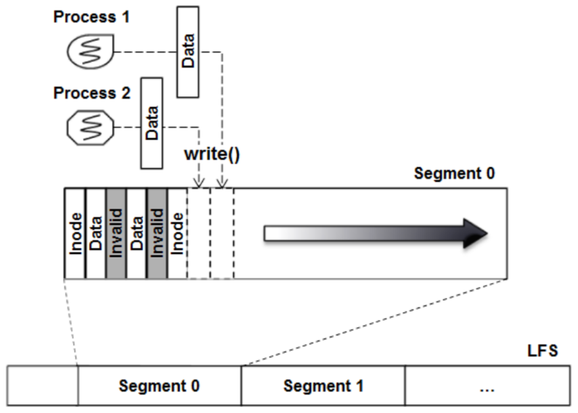


Figure 2. This figure shows the layout of the log- structured file system.[1]

The disk is divided into segments and only one segment can be active at one time in the log-structured file system. Every individual segment has a header and it is called a summary block. This block contains a pointer to the next summary block and links to the long chain. This long chain that a log-structured file system treats as a linear log. The segments are not joined with each other on the disk and because of this larger segment sizes (between 384Kb and 1Mb) are recommended.

LFS does this through a time-honored method called write buffering. LFS maintains track of changes in memory prior to writing to the disk; when it has enough updates, it writes them all at once to the disk and makes effective use of the disk possible. The name segment denotes the substantial number of updates that LFS writes all at once. Even though this segment is overused in computer systems, here it simply refers to the biggest chunk that LFS employs to aggregate writes. As a result, while writing to the disk, LFS buffers updates in a segment in memory before publishing the segment in its entirety to the disk. These writes will be effective if the section is big enough.

The following is an illustration of how LFS buffers two sets of updates into a single tiny segment; actual segments are bigger (a few MB). The first update involves writing four blocks to file j, while the second involves adding one block to file k. The full section of seven blocks is then simultaneously committed to the disk by LFS. The arrangement of these blocks is as follows.

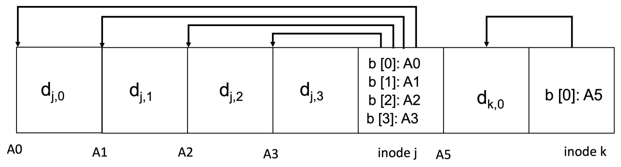


Figure 3.

3.1.2 BUFFERING TIME

Consequently, the question of how many updates LFS should buffer before writing to disk arises. The disk itself determines the answer, particularly how high the positioning overhead is in relation to the transfer rate.

Consider the scenario where positioning, such as rotation and search overheads, occur before each writes and require around T position seconds. Assume further that R MB/s is the disk transfer rate. On a disk like that, how much should LFS buffer before writing?

The best way to approach this is to think of the positioning cost as a set overhead that you pay each time you write. What volume of writing is required to find a way to reduce that cost? The better (naturally) and closer you get to reaching its peak bandwidth, the more you write. We can assume of getting a specific response that we are writing D MB. The positioning time Tp plus the transfer time D (D/R) equals the time required to write down this chunk of data (T), or T = Tp + D /R.

Thus, the effective rate of writing (Reff), which is simply the quantity of data written divided by the total length of time to write it, is given by the formula Reff = D /T = D /(Tp+ D /R).

Here, we aim to narrow the gap between the effective rate and the R rate. A typical F may be 0.9, or 90 percent of the R rate, since 0 F 1. Specifically, we want the effective rate to be some fraction of F of the R rate. This indicates that in mathematical terms, we need Reff = F* R. [1]

3.1.3 Inode

On the disk, inodes exist in a static form. Disk inode contains owners' information, file type, file access permission, access time, number of links, file size, and an array of disk blocks. The in-core inode is an additional field of the disk inode. It contains the status of inodes, the device number, the inode number, points to the inode, and a reference count.

There is a noticeable difference between a buffer header and an in-core inode reference count. When a process allocates an inode the reference count will increase. If the reference count is 0 then it will be in a free list and if the reference count is greater than 0 that means that processes are accessing inodes. When processes are accessing inodes it means inodes are in the hash queue.

To access the inode we use iget algorithm, which will allocate an in-core copy of an inode. If inodes are found in the free list it will allocate an inode and reads the disk copy into the in-core inode. As we mentioned above the in-core contains some fields, which means this in-core copy knows the inode number and device number.

According to the number of inodes fit on one disk block, this algorithm will calculate the logical block number. Here is the algorithm for logical block numbers:

block number=((inode number - 1) / number of inodes per block) + start block of inode list

3.1.4 How to find Inode

We must notice how to discover an inode in a standard UNIX file system to better understand how we find inodes in LFS. Finding inodes is simple in a normal file system like FFS or even the previous UNIX file system since they are arranged in an array and put in fixed places on a disk. The ancient UNIX file system, for instance, stores each inode at a set location on the disk. Therefore, given an inode number and the start address, you may identify a specific inode by calculating its precise disk address by multiplying the inode number by the size of an inode and adding that value to the on-disk array start address; this is known as array-based indexing is quick and simple. Because FFS divides the inode table into sections and sets a group of inodes within each cylinder group, finding an inode given an inode number is just a little more difficult.

Therefore, one must be aware of the start addresses and size of each block of inodes. Calculations after that are comparable and simple. It is more challenging in LFS because we were able to disperse the inodes throughout the whole disk. To address this, the creators of LFS created a data structure known as the inode map that adds a layer of indirection between inode numbers and the inodes (IMAP). An inode number is entered into the IMAP structure, which outputs the disk address of the inodes most current version. So, it seems sensible that it would frequently be implemented as a straightforward array with 4 bytes (a disk pointer) for each item. The IMAP is always updated with an inodes new position whenever one is written to disk.

So, the IMAP must be maintained persistent (that is, written to disk); and doing this enables LFS to remember the locations of inodes despite crashes and function as intended. Here we know the location of IMAP might reside on a set area of the disk. Because of the frequent changing of the inode, performance would suffer if modifications to file structures weren't followed by writes to the IMAP (i.e., there would be more disk seeks, between each update and the fixed location of the IMAP). Instead, LFS writes new data immediately adjacent to where it is writing portions of the inode map. LFS puts the new data block, its inode, and a portion of the inode map all together onto the disk when appending a data block to a file k, as seen below:

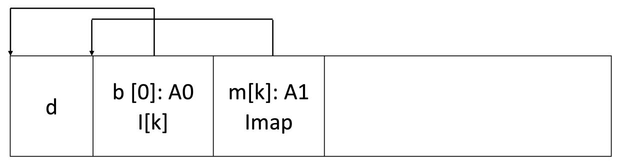


Figure 4.

The portion of the IMAP array seen above is contained in the block labeled IMAP, which informs LFS that the inode k is at disk address A1 and that its data block D is located at address A0 through the inode.

Consider the size of the inode map if there were 100,000 inodes.

100000 * 4 = 400000 bytes, or 390 KB, is the size of the inode map.

Inode maps can therefore be entirely cached in memory due to their modest size. There is a duplicate of the inode map in each segment on the disk.[1]

3.1.5 Where to put the inode map on disk?

A specific area of the disk may be used to mount an inode map. But because it is updated regularly, we must also update the inode map each time a segment is altered. Thus, there would be more disk seeks, which might have an influence on performance. Please be aware that we sync segments to disk before they are filled with updated inodes or data. Since the segments are only partially changed, updating the inode map with each partial write might affect speed. Therefore, we maintain the inode map for each segment. [1]

3.1.6 Find the inode-map

The address of the most recent inode-map is stored by LFS in a predefined location on the hard drive known as the checkpoint region (CR). Every time we create a new section, the CR needs to be updated. As a result, CR needs to be changed only seldom.[1]

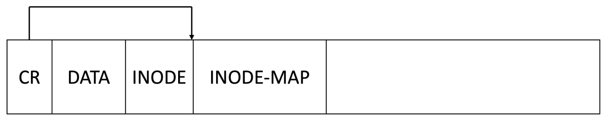


Figure 5.

The overall name of the file system is a log-structured file system since each segment in this case has a pointer to the following segment. Beginning with the CR, we may locate

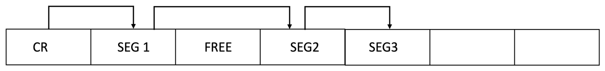


Figure 6.

the initial segment written to the disk, follow the chain to determine what modifications were made to the file system, and ultimately locate the most recent segment.[1]

3.1.7 Segment Cleaning

The segment cleaning part is critical in the log-structured file system. The log-structured file system will work fast only if there are more segments available is the disk. Once the files get modified the use of segments drops over time. Almost all the segments are not found totally empty. Segment cleaning is also tricky. Normal log use is dominated by writes and minimizes seeks. Cleaning is dominated by reads and requires many seeks.

The process of copying live data out of a segment is called segment cleaning. We have used a basic approach to clean the segment. We are trying to first read several numbers of the segment into memory and identify the current data after this we write the current data into a smaller number of the clean segment. Once this process is done, the segments will be marked as clean and can be used for new data. In our implementation we use 0 and 1, 0 is for the segment that is clean and 1 is for the segment that is not clean.

As part of segment cleaning, it must be possible to identify which blocks of each segment are live, so that they can be written out again. It must also be possible to identify the file to which each block belongs and the position of the block within the file; this information is needed to update the file's inode to point to the new location of the block. [1]

The summary block identifies each piece of information that is written in the segment; for example, for each file data block, the summary block contains the file number and block number for the block. Segments can contain multiple segment summary blocks when more than one log write is needed to fill the segment.

3.1.8 Segment Cleaning Policy

Segment cleaning policy issue can be addressed four policy.

1. Execution: Segment cleaning execution should be based on the prioritization of the task or it should be based on space available on the disk once it's exhausted.

2. Amount: As a segment cleaning provides an opportunity to clear up more space by rearranging the data which in turn will help to reorganize the data on the disk.

3. Priority: Segment cleaning choice should be based on the most fragmented data to the least fragmented data on the disk.

4. Grouping: there are a few different ways to enhance the locality of future reads like grouping all the files in the same directory into a single output segment and another way is by sorting the blocks when they were last modified or by sorting them based on their age.

In our implementation, we have used the first two policies where the segments are cleaned when the disk space is exhausted and the second is we are able to clean any number of dirty segments at once.

3.1.9 LFS : Crash Recovery

Log-structured file systems can recover from unexpected crashes by using the most recent checkpoint, which leaves them in a consistent state and removes any later-written data. Additionally, we introduce a function to recover nearly all the data that was written after a checkpoint.

Until now, LFS provides two options for regaining control after an unplanned accident. The easier option is to ignore any data that was written after the last checkpoint, which is assumed to be in a perfectly consistent state. In addition, we offer a roll-forward tool that allows you to recover as much data as you can, even if it was added after a checkpoint. If the segments checksums are accurate, this utility begins at the most recent checkpoint and proceeds along the chain of segments that have been written since then. By looking at the segment summary and file information records, all entities in each of these segments are found, and the necessary metadata is updated. The tool updates the relevant indirect block or inode whenever a data block is read. The inode table is updated to point to this copy of the inode if an inode is found. The segment usage table needs to be changed in both scenarios.

The utility must additionally handle inode and directory consistency. The link counter of an inode may not match the number of directory entries if a crash happens while only a portion of a directory action has been written to the disk, or such an item may refer to an inode that doesn't exist or even the erroneous inode. The fundamental issue is that the majority of directory actions touch multiple inodes, thus either all changes must be recovered during roll forward, or none at all. To combat the issue, BSD- LFS marks any partial segments that have unfinished directory activities and refuses to conduct roll-forward on them unless another segment comes after it and finishes the operations. For each directory operation, Sprite-LFS add a record to the log. Together, these documents make up a directory operation journal. Both file systems ensure that the relevant journal item comes before any affected inode or directory block in the log. Although this method makes roll-forward more challenging to construct, it permits the recovery of more data and places relatively few restrictions on the directory operations' implementation, allowing for simpler and quicker execution.

Pointers to all the on-disk IMAP components are contained in the on-disk checkpoint region.

1. In the log-structured file system, checkpoint regions are updated every 30 seconds.

2. After a crash, the checkpoint log-structured file system can be used to recreate the in-memory map.

3. To ensure that it will be automatically updated, the log-structured file system creates two copies of checkpoint regions in two separate areas of the disk.

4. LFS switches which place gets the current checkpoint.

5. When a checkpoint is interrupted, LFS writes a timestamp,the checkpoint is only valid if both timestamps match and pointers to the imap fragments are provided.

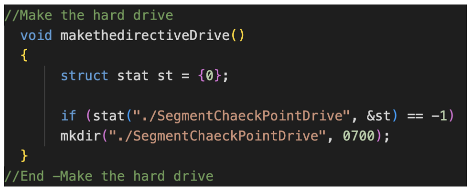
6. This method is effective, but we lose the final 30 seconds of data writing.

4 Implementation and Results

We tried to implement a simple log-structured file system that focuses on disk organization where the user will be able to get the text file from LINUX or MacOs and will be able to read, update and also delete the file from the log-structured file system. We have also implemented commands to test the Read, Write, Segment cleaning, Segment Summary of the active segment, Imap of active Segment, Delete file from the system, and Current active Segment. Here buffer cache will be handled by the host operating system memory management.

4.1 Drive Creation

The sizes and types of several disk data structures, such as blocks, and inodes, have been implemented. In continuation to the implementation, our file system is using a single LINUX directory named as "SegmentCheckPointDrive" for implementing of our log structure file system and to create this drive We have created a function that will create a drive with the name of "SegmentCheckPointDrive" as shown in the figure - 7.

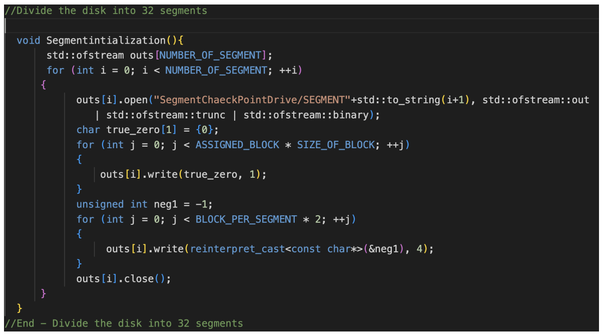


```
//Make the hard drive
  void makethedirectiveDrive()
  {
      struct stat st = {0};

      if (stat("./SegmentChaeckPointDrive", &st) == -1)
      mkdir("./SegmentChaeckPointDrive", 0700);
  }
//End -Make the hard drive
```

Figure 7. Creating Drive for log structure file system

4.2 Segment Creation

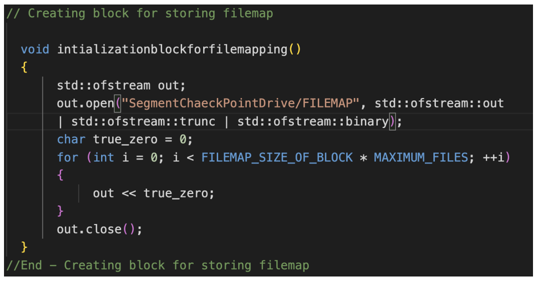
The Drive we created contains some segment files, as its back-end disk drive.The drive will consist of 32 segments, each of which can have up to 1MB of null data. The directory will be created SegmentCheckPointDrive and the segment files are called as SegmentCheckPointDrive/SEGMENT1, SegmentCheckPointDrive/SEGMENT2, SegmentCheckPointDrive/SEGMENT32.



```
//Divide the disk into 32 segments

  void Segmentintialization(){
      std::ofstream outs[NUMBER_OF_SEGMENT];
      for (int i = 0; i < NUMBER_OF_SEGMENT; ++i)
     {
          outs[i].open("SegmentChaeckPointDrive/SEGMENT"+std::to_string(i+1), std::ofstream::out
             | std::ofstream::trunc | std::ofstream::binary);
          char true_zero[1] = {0};
          for (int j = 0; j < ASSIGNED_BLOCK * SIZE_OF_BLOCK; ++j)
          {
              outs[i].write(true_zero, 1);
          }
          unsigned int neg1 = -1;
          for (int j = 0; j < BLOCK_PER_SEGMENT * 2; ++j)
          {
               outs[i].write(reinterpret_cast<const char*>(&neg1), 4);
          }
          outs[i].close();
     }
  }
//End - Divide the disk into 32 segments
```

Figure 8. Creating 32 segment inside drive

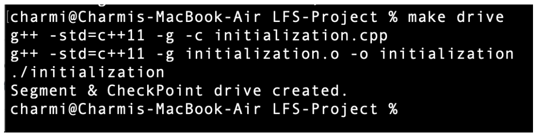
In the above figure-8, shows the functions that will create a file inside the "SegmentCheckPointDrive". In this drive, we will store the 32 Segments, File mapping, and Checkpoint Regions.



```
// Creating block for storing filemap

  void intializationblockforfilemapping()
  {
      std::ofstream out;
      out.open("SegmentChaeckPointDrive/FILEMAP", std::ofstream::out
      | std::ofstream::trunc | std::ofstream::binary);
      char true_zero = 0;
      for (int i = 0; i < FILEMAP_SIZE_OF_BLOCK * MAXIMUM_FILES; ++i)
      {
          out << true_zero;
      }
      out.close();
  }
//End - Creating block for storing filemap
```

Figure 9. Creating block for storing file-map inside drive

The below figure- 11 shows the output example, To execute and create the drive we have to use the ”make drive” command as shown in figure-10.



```
charmi@Charmis-MacBook-Air LFS-Project % make drive
g++ -std=c++11 -g -c initialization.cpp
g++ -std=c++11 -g initialization.o -o initialization
./initialization
Segment & CheckPoint drive created.
charmi@Charmis-MacBook-Air LFS-Project %
```

Figure 10. make drive command

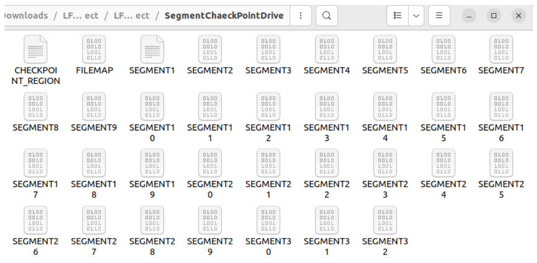
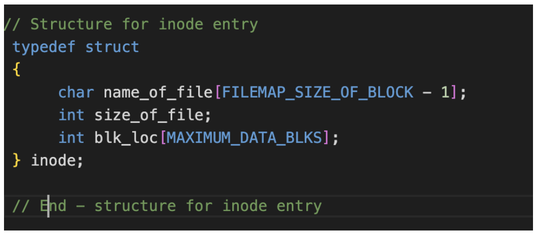


Figure 11. Drive output

4.3 Assigning Inode to the file

The file system that we are attempting to implement is a very basic UNIX file system. Inodes and data blocks are part of our file system. Each inode has the file’s string name, size, and 128 direct data block pointers rather than having 12 direct pointers as in inode implementation in sprite LFS. Indirect block pointers and double indirect block pointers are not supported by our file system. The maximum file size is 128K because there are only direct data blocks.

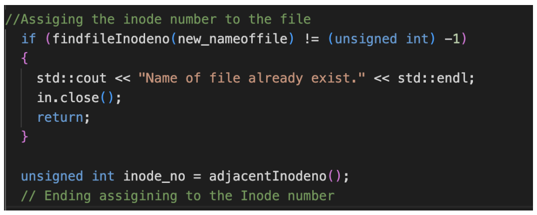


```
// Structure for inode entry
typedef struct
{
    char name_of_file[FILEMAP_SIZE_OF_BLOCK - 1];
    int size_of_file;
    int blk_loc[MAXIMUM_DATA_BLKS];
} inode;

// End - structure for inode entry
```

Figure 12. Inode Structure

So as shown in figure-12 we have created a structure in our implementation which will store the file name, file size and block location of the file buy using direct block pointers in the inode. In our implementation, there are no directories and links. We allocated inodes the same size as data blocks, which is 1k, even though they don’t need to be that huge.

We have created a function that will assign the inode number to each imported file.

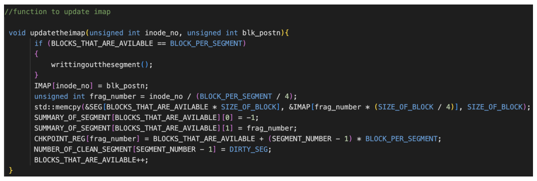


```
//Assiging the inode number to the file
  if (findfileInodeno(new_nameoffile) != (unsigned int) -1)
  {
    std::cout << "Name of file already exist." << std::endl;
    in.close();
    return;
  }

  unsigned int inode_no = adjacentInodeno();
  // Ending assigining to the Inode number
```

Figure 13. Function to assign Inode to imported file

4.4 Imap

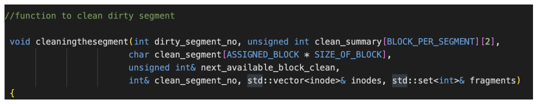
Our approach supports an IMAP structure with a home on the disk in the log (that is, inside the 32 segments), and we record a single checkpoint area with the imap’s position. The checkpoint region is stored in a separate file called CHECKPOINTREGION, which is located outside of any of the 32 segment blocks. The CHECKPOINTREGION file contains a list of block numbers that are used to store sections of the IMAP in the correct sequence. In 4-byte unsigned inits, both inodes and block numbers are stored. The Log-Structured File System has a file limit of 10K. As a result, the IMAP is saved in 80K bytes or 80 disk blocks. As a result, the checkpoint region is represented as a file of precisely 320 bytes, no more, no less. When our file system process boots up, it reads the checkpoint region and puts together anything else it needs.



```
//function to update imap

 void updatetheimap(unsigned int inode_no, unsigned int blk_postn){
        if (BLOCKS_THAT_ARE_AVILABLE == BLOCK_PER_SEGMENT)
        {
            writtingoutthesegment();
        }
        IMAP[inode_no] = blk_postn;
        unsigned int frag_number = inode_no / (BLOCK_PER_SEGMENT / 4);
        std::memcpy(&SEG[BLOCKS_THAT_ARE_AVILABLE * SIZE_OF_BLOCK], &IMAP[frag_number * (SIZE_OF_BLOCK / 4)], SIZE_OF_BLOCK);
        SUMMARY_OF_SEGMENT[BLOCKS_THAT_ARE_AVILABLE][0] = -1;
        SUMMARY_OF_SEGMENT[BLOCKS_THAT_ARE_AVILABLE][1] = frag_number;
        CHKPOINT_REG[frag_number] = BLOCKS_THAT_ARE_AVILABLE + (SEGMENT_NUMBER - 1) * BLOCK_PER_SEGMENT;
        NUMBER_OF_CLEAN_SEGMENT[SEGMENT_NUMBER - 1] = DIRTY_SEG;
        BLOCKS_THAT_ARE_AVILABLE++;
 }
```

Figure 14. Update imap

As a log-structured file system, data blocks, inodes, and blocks that contain parts of the IMAP are moving around the disk when changed and are written at the end of the log. We have implemented the segment cleaning process that reads segments and determines which of their block contain the live data by using segment summary blocks and inodes and writing those blocks into the same of the segments freeing up others. our file system keeps the list of the free segment and uses them as needed. The file system contains the matches of the size of a single segment that is 1MB and operates on blocks in that buffer until it fills up, at that point entire buffer overwrites one of the free segments.



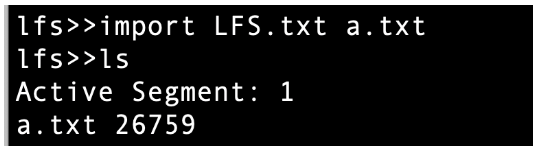
```
//function to clean dirty segment

 void cleaningthesegment(int dirty_segment_no, unsigned int clean_summary[BLOCK_PER_SEGMENT][2],
                         char clean_segment[ASSIGNED_BLOCK * SIZE_OF_BLOCK],
                         unsigned int& next_available_block_clean,
                         int& clean_segment_no, std::vector<inode>& inodes, std::set<int>& fragments)
 {
```

Figure 15. Segment Cleaning function

4.5 Commands

To test and experiment the results, we have created a list of functions where we can understand the log-structured file system.

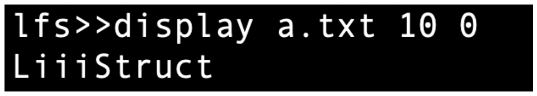
1. *import filename newfilename*
   - It will import the text file “filename” and save it under the new name “new filename” in log structure file system.
   - e.g import LFS.txt a_LFS.txt



```
lfs>>import LFS.txt a.txt
lfs>>ls
Active Segment: 1
a.txt 26759
```

Figure 16. Import command O/P

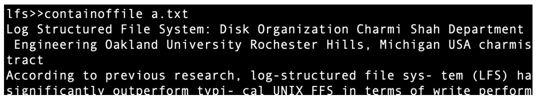
2. *display new_filename howmany start*
   - It will display the “how many” bytes of content of the new filename starting from byte that you want to “start” with.
   - e.g display a_a.txt 100 0



```
lfs>>display a.txt 10 0
LiiiStruct
```

Figure 17. Display the content

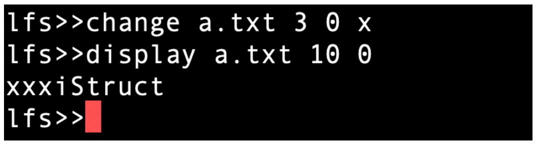
3. *importntimes filename new_filename howmany*
   - It will import ”filename” and name this file with “new filename” and will import this file for “number” how-many times you want that file.
   - e.g importntimes LFS.txt a.txt 6
4. *containoffile newfilename*
   - It will display the content of the “new filename” that you want to see.
   - e.g containoffile a_a.txt



```
lfs>>containoffile a.txt
Log Structured File System: Disk Organization Charmi Shah Department
 Engineering Oakland University Rochester Hills, Michigan USA charmis
tract
According to previous research, log-structured file sys- tem (LFS) ha
significantly outperform typi- cal UNIX FFS in terms of write perform
```

Figure 18. File contain O/P

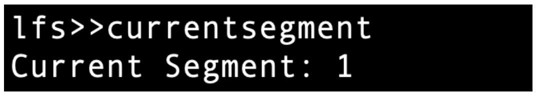
5. *ls*
   - It will list all the names and sizes of the files.
   - e.g ls
6. *change new_filename howmany-character start char-to-be-replaced*
   - It will write ” how many” copies of the character “char” into the file’s “new filename” beginning at byte ”start”. If this surpasses the original size of the file, the overwrite will grow the file.
   - e.g change a_a.txt 4 1 a



```
lfs>>change a.txt 3 0 x
lfs>>display a.txt 10 0
xxxiStruct
lfs>>
```

Figure 19. Overwrite the characters

7. *currentsegment*
   - It will display the currently active segment number.
   - e.g currentsegment



```
lfs>>currentsegment
Current Segment: 1
```

Figure 20. current segment

8. *segmentsummary howmany start*
   - It shows the active segment summary by displaying “how-many” unsigned ints of the segment summary starting from ”start”.
   - e.g segmentsummary 10 1
9. *deletefile filename*
   - It will delete the file with the name ”new filename” from log structure file system.
   - e.g deletefile a.txt
10. *clean 2*

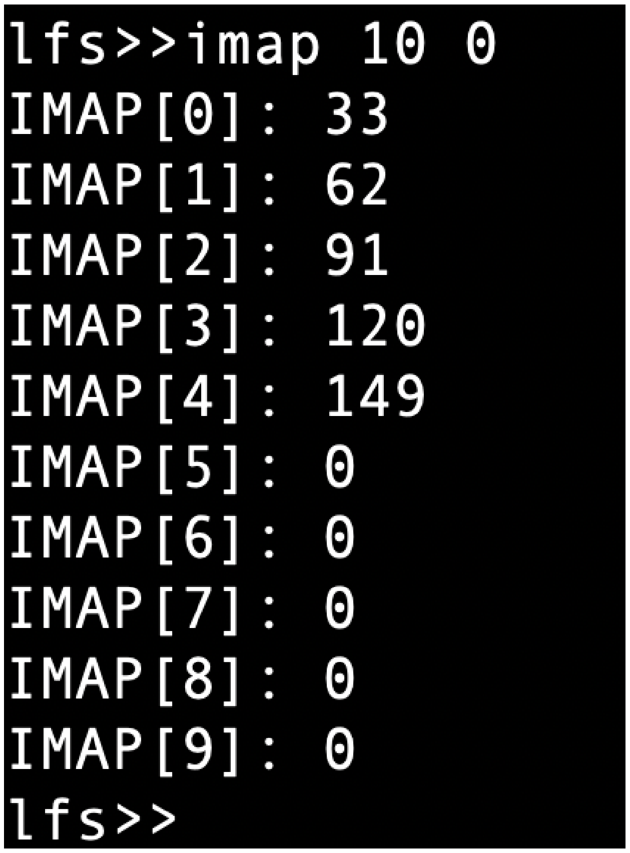
- It will take ”how-many” dirty segments you want to clean and it will clean the dirty segments from the disk.
- e.g clean

11. *cleansegments*

    - It will display the clean segments remaining on the disk.
    - e.g cleansegments

12. *imap howmany start*

    - It will display ”how-many” unsigned int’s from the IMAP of the active segment starting from ”start” that user specified through command.
    - e.g imap 4 1

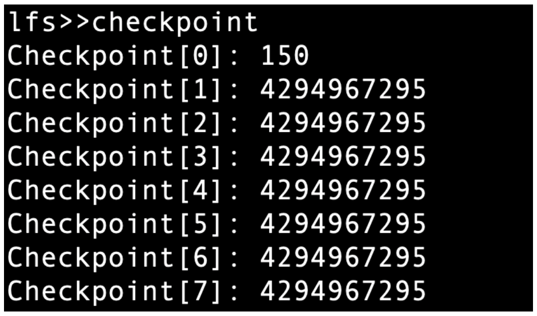


```
lfs>>imap 10 0
IMAP[0]: 33
IMAP[1]: 62
IMAP[2]: 91
IMAP[3]: 120
IMAP[4]: 149
IMAP[5]: 0
IMAP[6]: 0
IMAP[7]: 0
IMAP[8]: 0
IMAP[9]: 0
lfs>>
```

Figure 21. Display the content

13. *checkpoint*

    - It will display the contents of the checkpoint region that is stored on the disk.
    - e.g checkpoint



```
lfs>>checkpoint
Checkpoint[0]: 150
Checkpoint[1]: 4294967295
Checkpoint[2]: 4294967295
Checkpoint[3]: 4294967295
Checkpoint[4]: 4294967295
Checkpoint[5]: 4294967295
Checkpoint[6]: 4294967295
Checkpoint[7]: 4294967295
```

Figure 22. Display the content of checkpoint

14. *nextblock*

    - It will display the Next available block on the disk.
    - e.g nextblock

15. *exit*

    - It will write out cached memory to disk and exit the shell.
    - e.g exit

5. Feasible new Method or Solution

A feasible solution to solve this modern file system solution is a log structure file system. Aggregating tiny random writes into a single large page and inserting it in a cyclic log to copy them to the ”ultimate destination” later when the demand on the disk subsystem is lower, is the concept that eventually gave rise to the concept of ”log-structuring”. File data blocks, attributes, index blocks, directories, and almost all the additional information needed to maintain the file system are all included in the information sent to disk during the writing process.[1]

A log-structured file system transforms the numerous tiny synchronous random writes of standard file systems into huge concurrent sequential transfers that may utilize about 100 percent of the raw disk capacity for workloads that comprise several small files [1] like fast failure recovery, fast snapshots and remarkable findings in terms of the randomized writes pattern.

But unlike the cache-logging method, it only utilizes one disk version of the data. Log-structured file systems might potentially boost disk bandwidth utilization along with certain other additional attractive qualities.[5]

6. Future Research Plan

Overall, using a log-structured file system is a smart concept, but you should take caution as it is not suitable for all jobs and workloads. Log-structure file system comes with the following problem that needs to be resolved in near future, such as.

1. Sequential reading performance is unpredictable. It might not be as effective as traditional file systems (in case the data is written in the next sector). Additionally, it might be as terrible as random reading as opposed to sequential reading. Performance essentially relies on the kind and size of the workload at the time of writing.
2. The need for “Garbage Collection”.
3. The requirement for a lot of empty space.
4. Manage large and complex data structures on disk.
5. Work with flash memory in common formats, such as SSD and NVMe, has been inadequate.

7. Team Contributions

In this project, We are trying to test the log-structured file system to understand the disk organization. Both team member were responsible for certain aspects of the project and our duties were laid out clearly such as gathering the reference information, understanding the algorithm and implementation of the overall project. As the Operating System was a new topic for us, We equally weighed in terms of understanding the subject via weekly meetings and literature sharing. This project was a great opportunity for both of us as it helped us to understand one of the core principle of Operating System.

References

[1] Mendel Rosenblum. 1992. ”The design and implementation of a log-structured file system.” Ph.D. Dissertation. University of California at Berkeley, USA.

[2] H. Gwak, Y. Kang and D. Shin, ”Reducing garbage collection overhead of log-structured file systems with GC journaling,” 2015 International Symposium on Consumer Electronics (ISCE), 2015

[3] Jonggyu Park, Dong Hyun Kang, and Young Ik Eom. 2016. File Defragmentation Scheme for a Log-Structured File System.In Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys ’16). Association for Computing Machinery, New York, NY, USA, Article 19.

[4] Martin Jambor, Tomas Hruby, Jan Taus, Kuba Krchak, and Viliam Holub. 2007. Implementation of a Linux log-structured file system with a garbage collector. SIGOPS Oper. Syst. Rev. 41, 1 (January 2007)

[5] F. Douglis and J. Ousterhout, ”Log-structured file systems,” Digest of Papers. COMPCON Spring 89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage