Department for Mathematics & Computer Science
University of Southern Denmark

DM510 Operating Systems

Department for Mathematics & Computer Science

IMADA

Project 4: Log-Structured Filesystem

Torvald Johnson – tjohn16

April 15, 2017

Table of Contents

1 Introduction

The task at hand was implement a log-structured filesystem, based on a description of such a system by Tanenbaum and Woodhull, and using Linux FUSE. More specifically, a file system specified as an LFS, or log-structured filesystem, should store all data in memory until there is a need to write to the disk. At that time, pending writes buffered in memory should be collected and written as a single contiguous segment to the disk, at the end of the "log" of previous data. Importantly, a map of inodes, indexed by inode-number, should be maintained both on disk and in memory at all times, so that inodes can be easily located and thus, blocks as well can be found. Another important detail is the cleaner function, which must run periodically to discard old, unused data and compact the log.

2 Design

A very important design decision faced when beginning work on this file system, was that of how much data to allocate to the various parts of the system. It was necessary to decide how large blocks should be, how large segments should be, how many segments should exist in the filesystem, the maximum amount of inodes that should be able to exist on the system, and how big the inodes should be. I decided that things should be kept as simple as possible, due to the large and daunting nature of the project, so the decision was made to keep the entire "hard drive" size limited to 1MB. I divided this hard drive into 4 segments, meaning that each segment had a size of 262144 byes. This sizing was very nice, because it meant that I could specify blocks to be 1024 bytes, or 1 KB each. In this way, 256 blocks fit in each segment, and 4 segments fit in the entire hard drive.

Regarding the size and structure of the inodes, thinking back to the project description I realized it would be important to maintain a map of the inodes in the memory at all times, and write this map to the beginning of each segment to keep track of all data within the filesystem. As such, I wanted inodes to be relatively small. By limiting the filename held within an inode to 56 characters, I was able to keep the size of inodes down to 128 bytes. This meant that each block could hold 8 inodes, so by keeping 32 blocks "free" of data at the beginning of each segment, I would have space to write a map of 256 inodes to the filesystem in every segment.

The inodes were also designed in this system to hold an ID number corresponding to their location in the inode array, an int specifying the type of file or directory, the name and size of the file, time stamps for access and modification, as well as 8 datapointers and an indirect datapointer to an array of 32. This means that every inode could point to 40 blocks.

Another important aspect of this system, was going to be the ability to find any inode, and thus the data it points to, at any given time. As such, based upon the idea in the description of LFS systems holding a map of inodes, I decided that the best way to accomplish this would be with a recursive function to find the inode ID of any given path from the root. This way any inode in the system would be easily accessible from an array of all the inodes, once the filesystem had retrieved the ID of that inode.

Interestingly, the design of the deletion of files was relatively straightforward. Considering the necessity in the outline of an LFS for a cleaner function to remove any unattached data, the decision was made that the removal of any inode or data would simply amount to clearing that inode's pointer from the inode array, and clearing any pointer to that inode from its parent. After that, the cleaner function

when run periodically would preserve all intact inodes and data, while allowing old no longer used data to be overwritten.

Further considering the cleaner function, the decision was made that it would simply be run any time the file system moved to a new segment. In this way, the file system would remain relatively up-to-date at all times, without bogging down system resources by running the cleaner too often. This specific function and its implementation will be discussed in much greater detail later on in the implementation section of this report.

As far as the actual functionalities of this system, the decision was made to keep it moderately limited. Due to the relatively large size of the project, I thought it was more important to get the file system functioning on a bare level within the time limit, rather than to focus on many features. The system was designed to be able to list, create, delete, and rename files and directories, as well as open, read to, write from, and close files. I specifically decided that I would not focus on the functionality of retrieval of data after shutdown, and that every time you restart the file system, all data will be lost from the previous session.

3 Implementation

An important factor of this implementation to consider, is how the system will respond to a catastrophic event such as a sudden system crash or improper shut down. As previously noted, my system was not designed to take retrieval of data into account. It was also not designed to attempt to preserve data in the event of any sort of shut down, expected or otherwise. My system was designed to hold all data in memory at all times. It does write data to the “hard drive” file any time a segment is full and must increment, but it never reads from the hard drive file. As such, any data that has been written during this segment incrementation, will still be written to the hard drive file after a catastrophic event. However, all data that was saved only to memory will be lost. Furthermore, the system has been programmed to clear the hard drive file every time it initializes, so retrieval of data using this system would be impossible. If one desired to create a system which was more protected against power outages and such, of course the first step would be to design a system which could access saved data in the first place.

Moreover, when considering a log structured filesystem, one idea I had was to create a file on a fixed location in the physical hard drive, whose purpose is to hold temporary backup data and nothing else. This file would be separate from the log structured file structure, and would never have to worry about being moved or held in memory like the other files. It would have a specific maximum size, and would be completely overwritten upon reaching its capacity. The idea here, is that this backup file would be written with only the latest changes in the file system, and quickly become obsolete and replaced when newer changes were made. As such, upon an unexpected shutdown, a user may be able to access this backup file in hopes of obtaining their most recent changes which had not had a chance to be properly written to the hard drive yet. One might further increase the chance of saving valuable data by having several backup files, however this would mean increased space taken up by the backup files, thereby decreasing the useable space of the hard drive. Another drawback of this solution, is that these backup files I have described do not follow the log structured model, and so one would have to

implement a separate kind of file system just for saving and updating the backup files, as well as methods to periodically update and maintain them.

As previously specified, the removal of inodes is fairly straightforward. A pointer to every inode within this system is placed within an array of inodes, whenever an inode is created. When data is written to the file system, the relevant inode's datapointers will be set to point to that data. As can be seen in the included code excerpt, all that is required is to find the ID of the inode to be removed, and find the parent inode. Then with a simple for-loop, one can access and check every single one of the

```
  //get the parent inode
  memset(lfs_inode, 0, sizeof(inode));
  memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[lfs_parentInodeID] *
BLOCK_SIZE), sizeof(inode));

  //remove the inode datapointer from the parent inode
  int i;
  for (i=0; i<NUMBER_OF_DATAPOINTERS; i++) {
    if(lfs_inode->datapointer[i] == lfs_inodeID) {
      lfs_inode->datapointer[i] = -1;
    }
  }

  //check indirectDataPointers
  if(lfs_inode->indirectDataPointer != -1) {
    int *lfs_indirectPointersArray;
    lfs_indirectPointersArray = lfs_disk_in_memory + (lfs_inode->indirectDataPointer *
BLOCK_SIZE);

    int j;
    for(j=0; j<NUMBER_OF_INDIRECTPOINTERS; j++) {
      if(lfs_indirectPointersArray[j] == lfs_inodeID) {
        lfs_indirectPointersArray[j] = -1;
        lfs_inode->indirectDataPointer = lfs_insertData((char *)
lfs_indirectPointersArray, NUMBER_OF_INDIRECTPOINTERS * sizeof(int));
      }
    }
  }

  //insert the updated parent inode into array of inodes, and remove the original inode
from the array
  lfs_inodeArray[lfs_parentInodeID] = lfs_insertData((char *) lfs_inode, sizeof(inode));
  lfs_inodeArray[lfs_inodeID] = -1;

  free(lfs_inode);

  return 0;
}
```

Figure 3.1: Excerpt from lfs_removeInode()

parent inode's datapointers and indirect datapointers. If the inode to be removed is found, the pointer is cleared. At that point the updated parent is written to the disk in memory, and all that is left is to clear the inode to be removed's ID from the inode array. At that point, there are no references left in the

filysystem to this inode, and it will no longer be accessible to the user. The fact that it is still taking up space is not an issue in a log structured file system, because, as outlined in the design, once the cleaner has run and compacted the log, all new and relevant data will be at the front and all old data will be at the back, ready to be overwritten.

```
//check each inode in the old inode array
  int i;
  for (i=0; i<NUMBER_OF_INODES; i++) {
    if(lfs_inodeArray[i] != -1) {
      //found an inode. Get the inode
      memset(lfs_inode, 0, sizeof(inode));
      memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[i] * BLOCK_SIZE), sizeof(inode));

      if(lfs_inode->type == 0) {
        //the inode is a directory, copy it over.
        lfs_newinodeArray[i] = lfs_insertData((char *) lfs_inode, sizeof(inode));
      } else {
        //The inode is a file, check its datapointers.
        int j;
        for(j=0; j<NUMBER_OF_DATAPOINTERS; j++) {
          if(lfs_inode->datapointer[j] != -1) {
            //found data
            lfs_inode->datapointer[j] = lfs_insertData(lfs_disk_in_memory + (lfs_inode-
>datapointer[j] * BLOCK_SIZE), BLOCK_SIZE);
          }
        }
        //check indirectDataPointers
        if(lfs_inode->indirectDataPointer != -1) {
          //get the indirectDataPointers array
          int *lfs_indirectPointersArray;
          lfs_indirectPointersArray = lfs_disk_in_memory + (lfs_inode->indirectDataPointer *
BLOCK_SIZE);

          int k;
          for(k = 0; k<NUMBER_OF_INDIRECTPOINTERS; k++) {
            if (lfs_indirectPointersArray[k] != -1) {
              //found data
              lfs_indirectPointersArray[k] = lfs_insertData(lfs_disk_in_memory +
(lfs_indirectPointersArray[k] * BLOCK_SIZE), BLOCK_SIZE);
            }

          }
        //set the updated indirectDataPointers array
        lfs_inode->indirectDataPointer = lfs_insertData((char *) lfs_indirectPointersArray,
NUMBER_OF_INDIRECTPOINTERS * sizeof(int));

        free(lfs_indirectPointersArray);
        }
        //insert the updated inode into the new inode array
        lfs_newinodeArray[i] = lfs_insertData((char *) lfs_inode, sizeof(inode));
      }
    }
  }
  //overrwite the old inode array with the new updated inode array
  memset(lfs_inodeArray, 0, NUMBER_OF_INODES);
  memcpy(lfs_inodeArray, lfs_newinodeArray, NUMBER_OF_INODES);
```

Figure 3.2: Excerpt from the lfs_cleaner() function

Let's look a little deeper into the implementation of the cleaner, and what happens to data after it is removed. The unused datablocks are now ready to be cleaned up and rewritten with new data, as soon as the cleaner method moves the log forward. The cleaner design requires it to search through all old data, and only preserve data that is still intact and relevant. As such, for my implementation I wrote a function which creates a new empty inode array sized to the max number of inodes. The cleaner then loops through every index in the old inode array, checking first if there is a pointer to an inode set in that index. If so, it then checks every datapointer and indirect datapointer for that inode, and rewrites any data found to blocks on the current segment, while updating the pointers in the inode. Then when finished, the updated inode is inserted into the new inode array. After the cleaner has checked every single possible index in the old inode array, we can be certain that every remaining inode and all of its data has been moved up to the current segment, and we write the new inode array over the old one. In this way, all good data is moved up, and all old no-longer used data is left behind to be overwritten when the LFS loops back around.

Another interesting implementation I feel is worth a short mention, is the lfs_insertData() method. This method takes data, and writes it in the current lfs_block in the disk in memory, then returns that block's number. It's very useful, because it means that any kind of data can be inserted into the disk in memory, and the returned block number can be used as a pointer to that data. This is perfect, for example for inserting a new inode pointer into the inode array, and its parent's datapointer. Using lfs_insertData() the inode is written to the disk + (the current bock number * the size of a block), and the returned block number can be inserted into the inode array, providing an easy pointer to the inode. The same principle holds for setting up inode pointers when writing file data.

Besides that, the inode map is fairly interesting as well. Whenever an inode is created, that inode is inserted into a block using lfs_insertData, and its parent inodes datapointers and the inode array pointer are set to point to that block, which means that all inodes are connected. And since we know the root inode is always at index 0 in the array, if we have a path, we can always get the root and follow the path to the inode we're looking for, if it exists. Which is the main principle behind how the recursive lfs_findInodeID() function works.

4 Testing

There are five test files which have been created to test this filesystem. They are simple bash files, which run command line arguments to manipulate the file system and files within it. These tests are designed to ensure that the file system can function properly, carry out the objectives stated within the design section, and hold up to certain negative conditions. A video of these tests running will also be included, and this section of the report will go over the tests and their expected and actual output, providing a reference to the relevant time in the video for each test. Below, for your convenience, is a table of the tests listing their names, a short description of each one, and their timestamps in the included video.

Name:	lfsTest	lfsTest2	lfsTest3	lfsTest4	lfsTest5
Description:	Create a file and directory. List the files. Write to the file. Read the file. Create a file within the directory.	Create a file and directory. Delete the file and directory.	Create a file with a name that is 81 characters long. (longer than the max length of 56.)	Create 5 files, each with a large amount of text. Each file will be around 1KB. List the files. Read a file.	Create a file filled with the maximum amount of the file system, 1MB of data.
Result:	PASSED	PASSED	PARTIALLY PASSED	PASSED	FAILED
Timestamp:	0:03	0:22	0:34	0:49	1:14

Figure 4.1: Table of tests

The first test attempts to carry out a number of regular tasks a user might expect to perform inside of a filesystem. This test is described as follows: First a file will be created, named “newfile.” Then text will be written to that file, and read from that file. A directory called “newdir” will be created, and the contents of the root folder listed. The new directory will be entered, and “anotherfile” created inside of it. The contents will be listed. In order for this test to pass, all files and folders must be created, written to, read, and listed as expected. The test begins at 3 seconds into the video, and passes. The folder is shown in a window briefly at the end of this test, and the actions and results are output to the terminal.

Test 2 focuses on removing files from the file system. The description is as follows: A file and a directory will be created. They will be listed. The file and directory will be deleted. The contents of the empty root folder will be listed. In order for this test to pass, the file and folder must appear when created, and the root folder must be empty at the end of the test. The test begins at 22 seconds, and passes.

Test 3 is designed to see how the file system will cope with a filename that is longer than the specified maximum filename size. In this test: A file with a name 81 characters long will be created, and the contents of the root folder will be listed. In order for this test to pass, the file should be created, but should only have 56 characters in its name as specified in the file system design. The remaining characters will be lost. This test begins at 34 seconds, and it partially passes. The file was created and had the correct amount of characters in its name. However, when the test program attempted to create the file, the system outputted an error message specifying “cannot touch” followed by the original too-long filename. I believe that this is perhaps because the filesystem attempted to call lfs_getattr() on the original file path, and since the file name was changed during creation, the file could not be found. Unfortunately I haven’t had the time to debug this small issue.

Test 4 is about the creation of multiple semi-large files with a long string of text. In this test: 5 files, each 1KB in size, will be created and written to one after another with a 1 second delay between creation. The contents will then be listed, and one file will be read. For this test to pass, all files must be created and contain the correct text, a long string of “hellohellohello…”. The test takes place at 49 seconds, and passes.

Test 5 is designed to see how the file system holds up under pressure, find out if the file system can carry a single large file as big as the whole hard drive, and to test the ability of the file system to write a new segment and run the cleaner function. In this test: A file exactly 1MB in size will be created extremely quickly, filled with only null characters. The contents of the root folder will then be listed. In order for this test to pass, the file system must continue to work correctly and should not output any errors. This test takes place at 1:14 minutes, and it fails. The file system outputs a "segmentation fault" error, and the contents of the root directory cannot be listed. The file system ceases to work as expected. There are a few reasons for which I believe this happens. The number one reason is that a single inode only has 8 datapointers and 32 indirect pointers, making a total of 40 pointers that can point to blocks. This means that in this filesystem, a single file can only be up to 40KB (one block = 1KB) large. As such, a 1MB file could never be held in this filesystem under the current structure. There may, however, be other problems that I haven't had time to look into. For example, I have been unable to verify if my cleaner is working correctly, and there may be some problems once the file system actually loops back to the beginning and starts to rewrite old data. I haven't had the time to do any further testing on this matter. I believe that my design for the cleaner is sound, however I understand that I am very new to the area of filesystems and I can't count out the possibility that I may have overlooked some small issues in my implementation.

All in all, I am quite satisfied with the performance of the filesystem in these tests. I knew that test 5 would fail from the beginning, as previously stated, because my filesystem isn't designed to hold 1MB large files. Other than that, my filesystem can clearly create, delete, open, close, read, write, and rename files and folders.

6 Conclusion

This has been a very difficult project for me, and looking back I have incredibly mixed feelings about it. On the one hand, I am overjoyed that I was able to take a complex concept like the underlying mechanics of a filesystem in an operating system, and implement it, and see it working (although I know it is easier using FUSE and working in userspace, like we did.) On the other hand, this was very tough, I felt that I did not know anything at all about it going in, and throughout the project I was forced to work so rapidly and simply take and apply any steps, hints, and guidance that I found online, through other students, and through Troels (who was extremely helpful and patient with all of the students) as quick as possible just so that I could finish in time, I feel that I have missed out on an amount of knowledge I could have gained. That's not to say that I haven't learned anything, I absolutely have learned a lot about file systems and log structured file systems in particular, but I know there's so much more that I blew past along the way. However, I absolutely appreciate a challenging project like this, and I understand how pushing myself to the limits can open doors for me to learn more than I otherwise would have, so I do not regret my time spent on this project. However, even know as I frantically hurry to finish this report 1 hour before the deadline, I can't help but wish we had more time…

The project itself I feel was a success. The filesystem can list, create, delete, and rename files and directories, as well as open, read to, write from, and close files, just as specified in the design and in the project description. As such, I feel that it meets the defined requirements for this project. However, I can clearly see areas that my file system could and *should* be improved, and I simply haven't had the

time or knowledge to work on or improve them yet. Knowing what I know now, though, I am far better equipped for a project like this than I was just a month ago when I started, and that certainly means that this project has accomplished what I wanted it to.

7 Appendices

7.1 Appendix I: Source code for lfs.c

```
//INCLUDE

#include <fuse.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libgen.h>
#include <fcntl.h>
#include <time.h>
#include <utime.h>

//DEFINE

#define SEGMENT_SIZE 262144             //segment is 1/4 of 1MB. 256 blocks
per segment.
#define NUMBER_OF_SEGMENTS 4            //4 segments makes 1MB
#define BLOCK_SIZE 1024                 //each block is 1kb
#define NUMBER_OF_INODES 256            //we can use 32 blocks of each
segment to hold inodes.
#define MAX_LENGTH 56                   //max length to limit size of inodes
to 128 bytes
#define NUMBER_OF_DATAPOINTERS 8
#define NUMBER_OF_INDIRECTPOINTERS 32
#define HARDDISK "/home/tjohn16/dm510/project4/harddisk.txt"

//STRUCT INODE

typedef struct inode {
  int ID;                                   //ID number, also the number in
the inode array
  int type;                                 //0 = directory, 1 = file
  char name[MAX_LENGTH];                    //directory or file name, max
length is 56
  int size;                                 //filesize
  time_t modify;                            //modification time stamp
  time_t access;                            //access time stamp
  int datapointer[NUMBER_OF_DATAPOINTERS];  //8 datapointers
```

```
  int indirectDataPointer;                  //indirect data pointer
} inode;                                    //inode size is 128 bytes

//DEFINE METHODS

int lfs_init(void);
int lfs_getattr( const char *, struct stat * );
int lfs_readdir( const char *, void *, fuse_fill_dir_t, off_t, struct
fuse_file_info * );
int lfs_mknod(const char *, mode_t, dev_t);
int lfs_mkdir(const char *, mode_t);
int lfs_unlink(const char *);
int lfs_rmdir(const char *);
int lfs_rename(const char* from, const char* to);
int lfs_truncate(const char *path, off_t size);
int lfs_open( const char *, struct fuse_file_info * );
int lfs_read( const char *, char *, size_t, off_t, struct fuse_file_info * );
int lfs_write (const char *, const char *, size_t, off_t, struct
fuse_file_info *);
int lfs_release(const char *path, struct fuse_file_info *fi);
int lfs_findInodeID(char *);
int lfs_createInode(char *, int);
int lfs_removeInode(char *);
int lfs_insertData(char *, int);
int lfs_cleaner(void);
int lfs_write_segment(int, const char *);
int lfs_utime(const char *, struct utimbuf *);

//STRUCT FUSE OPERATIONS

static struct fuse_operations lfs_oper = {
      .getattr    = lfs_getattr,          //get attribute
      .readdir    = lfs_readdir,          //read directory
      .mknod = lfs_mknod,               //make file
      .mkdir = lfs_mkdir,               //make directory
      .unlink = lfs_unlink,             //remove file
      .rmdir = lfs_rmdir,               //remove directory
      .rename = lfs_rename,             //change filename
      .truncate = lfs_truncate,         //change filesize
      .open = lfs_open,                 //open file
      .read = lfs_read,                 //read file
      .write = lfs_write,               //write file
      .release = lfs_release,           //release file
      .utime = lfs_utime                //change access time
};

//GLOBAL VARIABLES

int *lfs_inodeArray;
void *lfs_disk_in_memory;
int lfs_segment;
int lfs_block;

//INIT METHOD

int lfs_init(void){
  printf("///  INITIALIZING FILE SYSTEM  ///\n");
```

```
  //allocate memory for the array of inodes
  lfs_inodeArray = malloc(NUMBER_OF_INODES * sizeof(inode));
  //allocate 1mb of memory for the disk
  lfs_disk_in_memory = malloc(NUMBER_OF_SEGMENTS * SEGMENT_SIZE);

  //create harddisk
  char harddisk[NUMBER_OF_SEGMENTS*SEGMENT_SIZE];
  int h;
  for (h=0; h<(NUMBER_OF_SEGMENTS*SEGMENT_SIZE); h++) {
    harddisk[h] = '0';
  }

  int file;
  file = creat(HARDDISK, 0777);
  write(file, harddisk, (NUMBER_OF_SEGMENTS*SEGMENT_SIZE));

  //set current block and segment
  lfs_segment = 0;
  lfs_block = 32;

  //initialize array of inodes
  int i;
  for(i=0; i<NUMBER_OF_INODES; i++) {
    lfs_inodeArray[i]=-1;
  }

  //create root inode
  inode *lfs_root;
  lfs_root = malloc(sizeof(inode));

  lfs_root->ID = 0;
  lfs_root->type = 0;
  memset(lfs_root->name, 0, 2);
  memcpy(lfs_root->name, "/", 2);
  lfs_root->size = 0;
  lfs_root->modify = time(NULL);
  lfs_root->access = time(NULL);

  int j;
  for(j=0; j<NUMBER_OF_DATAPOINTERS; j++) {
    lfs_root->datapointer[j] = -1;
  }

  lfs_root->indirectDataPointer = -1;

  //insert root inode into the disk in memory
  memset(lfs_disk_in_memory + (lfs_block*BLOCK_SIZE), 0, sizeof(inode));
  memcpy(lfs_disk_in_memory + (lfs_block*BLOCK_SIZE), (char *) lfs_root,
sizeof(inode));
  lfs_inodeArray[0] = lfs_block;

  //go to next block
  lfs_block++;

  printf("FILE SYSTEM INITIALIZED\n");
```

```
  return 0;
}

//GET ATTRIBUTE METHOD

int lfs_getattr( const char *path, struct stat *stbuf ) {
      printf("getattr method called\n");

      memset(stbuf, 0, sizeof(struct stat));

      //create an inode pointer
  inode *lfs_inode;
  lfs_inode = malloc(sizeof(inode));

  //find the inode
  int lfs_inodeID;
  lfs_inodeID = lfs_findInodeID(path);

  //if there is no inode
  if(lfs_inodeID == -1) {
    free(lfs_inode);

    return -ENOENT;
  } else {
    //get the inode
    memset(lfs_inode, 0, sizeof(inode));
    memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[lfs_inodeID] *
BLOCK_SIZE), sizeof(inode));

    //if the inode is a directory
    if(lfs_inode->type == 0) {
      stbuf->st_mode = S_IFDIR | 0777;
      stbuf->st_atime = lfs_inode->access;
            stbuf->st_mtime = lfs_inode->modify;

              return 0;
      } else if (lfs_inode->type == 1){
        stbuf->st_mode = S_IFREG | 0777;
        stbuf->st_atime = lfs_inode->access;
            stbuf->st_mtime = lfs_inode->modify;
              stbuf->st_size = lfs_inode->size;

              free(lfs_inode);

              return 0;
      }
  }
  return 0;
}

//READ DIRECTORY METHOD

int lfs_readdir( const char *path, void *buf, fuse_fill_dir_t filler, off_t
offset, struct fuse_file_info *fi ) {
      printf("readdir method called\n");
```

```
  filler(buf, ".", NULL, 0);
      filler(buf, "..", NULL, 0);

  int lfs_inodeID;
  lfs_inodeID = lfs_findInodeID(path);

  //get the inode
  inode *lfs_inode;
  lfs_inode = malloc(sizeof(inode));
  memset(lfs_inode, 0, sizeof(inode));
  memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[lfs_inodeID] *
BLOCK_SIZE), sizeof(inode));

  //read through the inode's datapointers
  int i;
  for(i=0; i<NUMBER_OF_DATAPOINTERS; i++) {
    if(lfs_inode->datapointer[i] != -1) {
      filler(buf, ((inode*)(lfs_disk_in_memory + (lfs_inodeArray[lfs_inode-
>datapointer[i]] * BLOCK_SIZE)))->name, NULL, 0);
    }
  }

  //read through the inode's indirectDataPointers
  if(lfs_inode->indirectDataPointer != -1) {
    int *lfs_indirectPointersArray;
    lfs_indirectPointersArray = malloc(NUMBER_OF_INDIRECTPOINTERS *
sizeof(int));

    //get the array of indirectDataPointers from the disk in memory
    memset(lfs_indirectPointersArray, 0, NUMBER_OF_INDIRECTPOINTERS *
sizeof(int));
    memcpy(lfs_indirectPointersArray, lfs_disk_in_memory + (lfs_inode-
>indirectDataPointer * BLOCK_SIZE), NUMBER_OF_INDIRECTPOINTERS *
sizeof(int));

    //checking each indirectDataPointer
    int j;
    for(j=0; j<NUMBER_OF_INDIRECTPOINTERS; j++) {
      if(lfs_indirectPointersArray[j] != -1) {
        filler(buf, ((inode*)(lfs_disk_in_memory +
(lfs_inodeArray[lfs_indirectPointersArray[j]] * BLOCK_SIZE)))->name, NULL,
0);
      }
    }
    free(lfs_indirectPointersArray);
  }
  free(lfs_inode);

      return 0;
}

//MKNOD METHOD

int lfs_mknod(const char * path, mode_t mode, dev_t dev) {
  int res;
  printf("mknod method called\n");
```

```
  //create the inode
  res = lfs_createInode(path, 1);

  return res;
}

//MKDIR METHOD

int lfs_mkdir(const char * path, mode_t mode) {
  int res;
  printf("mkdir method called\n");

  //create the inode
  res = lfs_createInode(path, 0);

  return res;
}

//UNLINK METHOD

int lfs_unlink(const char * path) {
  int res;
  printf("unlink method called\n");

  //remove the inode
  res = lfs_removeInode(path);

  return res;
}

//RMDIR METHOD

int lfs_rmdir(const char * path) {
  int res;
  printf("rmdir method called\n");

  //remove the inode
  res = lfs_removeInode(path);

  return res;
}

//RENAME METHOD

int lfs_rename(const char* from, const char* to) {
  printf("rename method called\n");

  //create an inode pointer
  inode *lfs_inode;
  lfs_inode = malloc(sizeof(inode));

  //find the inode
  int lfs_inodeID;
  lfs_inodeID = lfs_findInodeID(from);

  //get the inode
  memset(lfs_inode, 0, sizeof(inode));
```

```
  memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[lfs_inodeID] *
BLOCK_SIZE), sizeof(inode));

  //set the inode name
  if(strlen(basename(to)) < MAX_LENGTH) {
    memset(lfs_inode->name, 0, strlen(basename(from)));
    memcpy(lfs_inode->name, basename(to), strlen(basename(to)));
  }
  else {
    memset(lfs_inode->name, 0, strlen(basename(from)));
    memcpy(lfs_inode->name, basename(to), MAX_LENGTH);
  }

  //insert the inode into the inode array
  lfs_inodeArray[lfs_inodeID] = lfs_insertData((char *) lfs_inode,
sizeof(inode));

  free(lfs_inode);
  return 0;
}

//TRUNCATE METHOD

int lfs_truncate(const char *path, off_t size) {
  int res;
  printf("truncate method called\n");

  //create an inode pointer
  inode *lfs_inode;
  lfs_inode = malloc(sizeof(inode));

  //find the inode
  int lfs_inodeID;
  lfs_inodeID = lfs_findInodeID(path);

  //get the inode
  memset(lfs_inode, 0, sizeof(inode));
  memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[lfs_inodeID] *
BLOCK_SIZE), sizeof(inode));

  //set new size
  lfs_inode->size = size;
  lfs_inodeArray[lfs_inodeID] = lfs_insertData((char *) lfs_inode,
sizeof(inode));

  free(lfs_inode);

  return 0;
}

//OPEN METHOD

int lfs_open( const char *path, struct fuse_file_info *fi ) {
  int res;
  printf("open method called\n");

  //make sure inode exists
```

```
  res = lfs_findInodeID(path);
  if(res == -1) {
    return -ENOENT;
  }

      return 0;
}

//READ METHOD

int lfs_read( const char *path, char *buf, size_t size, off_t offset, struct
fuse_file_info *fi ) {
  int remain;
  printf("read method called\n");

  //create an inode pointer
  inode *lfs_inode;
  lfs_inode = malloc(sizeof(inode));

  //find the inode
  int lfs_inodeID;
  lfs_inodeID = lfs_findInodeID(path);

  //get the inode
  memset(lfs_inode, 0, sizeof(inode));
  memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[lfs_inodeID] *
BLOCK_SIZE), sizeof(inode));

  remain = lfs_inode->size;

  //read data
  int i=0;
  while(remain > BLOCK_SIZE && i<NUMBER_OF_DATAPOINTERS) {
    memset(buf + (i*BLOCK_SIZE), 0, BLOCK_SIZE);
    memcpy(buf + (i*BLOCK_SIZE), lfs_disk_in_memory + (lfs_inode-
>datapointer[i] * BLOCK_SIZE), BLOCK_SIZE);

    remain -= BLOCK_SIZE;
    i++;
  }

  if( i < NUMBER_OF_DATAPOINTERS) {
    //remaining data is within datapointers
    memset(buf + (i*BLOCK_SIZE), 0, remain);
    memcpy(buf + (i*BLOCK_SIZE), lfs_disk_in_memory + (lfs_inode-
>datapointer[i] * BLOCK_SIZE), remain);

    remain = lfs_inode->size;

    free(lfs_inode);

    return remain;
  }

  int *lfs_indirectPointersArray;
  lfs_indirectPointersArray = (int *)(lfs_disk_in_memory + (lfs_inode-
>indirectDataPointer * BLOCK_SIZE));
```

```
  int l = 0;
  while(remain > BLOCK_SIZE) {
    memset(buf + (i*BLOCK_SIZE), 0, BLOCK_SIZE);
    memcpy(buf + (i*BLOCK_SIZE), lfs_disk_in_memory +
(lfs_indirectPointersArray[l] * BLOCK_SIZE), BLOCK_SIZE);
    remain -= BLOCK_SIZE;

    i++;
    l++;
  }

  memset(buf + (i*BLOCK_SIZE), 0, remain);
  memcpy(buf + (i*BLOCK_SIZE), lfs_disk_in_memory +
(lfs_indirectPointersArray[l] * BLOCK_SIZE), remain);

  remain = lfs_inode->size;

      free(lfs_inode);

      return remain;
}

//WRITE METHOD

int lfs_write(const char * path, const char * buf, size_t size, off_t offset,
struct fuse_file_info * fi) {
  int remain;
  printf("write method called\n");

  //create an inode pointer
  inode *lfs_inode;
  lfs_inode = malloc(sizeof(inode));

  //find the inode
  int lfs_inodeID;
  lfs_inodeID = lfs_findInodeID(path);

  //get the inode
  memset(lfs_inode, 0, sizeof(inode));
  memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[lfs_inodeID] *
BLOCK_SIZE), sizeof(inode));

  remain = size;
  int i = 0;
  while(remain > BLOCK_SIZE && i < NUMBER_OF_DATAPOINTERS) {
    lfs_inode->datapointer[i] = lfs_insertData((char *) buf + (i*BLOCK_SIZE),
BLOCK_SIZE);
    remain -= BLOCK_SIZE;
    i++;
    }

    if(i < NUMBER_OF_DATAPOINTERS) {
      //write remaining data to datapointer
      lfs_inode-> datapointer[i] = lfs_insertData((char *) buf +
(i*BLOCK_SIZE), remain);
    } else {
```

```
      //more than one block of data remaining, write to indirectDataPointers
      int *lfs_indirectPointersArray = malloc(NUMBER_OF_INDIRECTPOINTERS *
sizeof(int));

      int l;
      for(l=0; l<NUMBER_OF_INDIRECTPOINTERS; l++) {
        lfs_indirectPointersArray[l] = -1;
      }
      int k = 0;
      while (remain > BLOCK_SIZE && k < NUMBER_OF_INDIRECTPOINTERS) {
        lfs_indirectPointersArray[k] = lfs_insertData((char *) buf +
(i*BLOCK_SIZE), BLOCK_SIZE);
        remain -= BLOCK_SIZE;
        i++;
        k++;
      }
      if(k < NUMBER_OF_INDIRECTPOINTERS) {
        lfs_indirectPointersArray[k] = lfs_insertData((char *) buf +
(i*BLOCK_SIZE), remain);
        lfs_inode->indirectDataPointer = lfs_insertData((char *)
lfs_indirectPointersArray, NUMBER_OF_INDIRECTPOINTERS * sizeof(int));
      }
      free(lfs_indirectPointersArray);
    }

  lfs_inode->size = size;
  lfs_inodeArray[lfs_inodeID] = lfs_insertData((char *) lfs_inode,
sizeof(inode));

  free(lfs_inode);

  return size;
}


//RELEASE METHOD

int lfs_release(const char *path, struct fuse_file_info *fi) {
  int res;
      printf("release method called\n");

  //make sure inode exists
  res = lfs_findInodeID(path);
  if(res == -1) {
    return -ENOENT;
  }

      return 0;
}

//FIND INODE ID METHOD

int lfs_findInodeID(char *path) {
  int res = 0;
  printf("findInodeID method called\n");

  //check if the given path is root
```

```
  if(strcmp("/", path) == 0) {
    //the root will always be at index 0
    return 0;
  }

  //create inode pointer
  inode *lfs_inode;
  lfs_inode = malloc(sizeof(inode));

  //if the given path is not root, we will recursively find the parent
  char *lfs_path;
  lfs_path = malloc(strlen(path)+1);
  memset(lfs_path, 0, strlen(path)+1);
  memcpy(lfs_path, path, strlen(path)+1);

  //find the parent inode
  int lfs_inodeID;
  lfs_inodeID = lfs_findInodeID(dirname(lfs_path));

  //get the parent inode
  memset(lfs_inode, 0, sizeof(inode));
  memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[lfs_inodeID] *
BLOCK_SIZE), sizeof(inode));

  //set the path back to the inode we're searching for
  memset(lfs_path, 0, strlen(path)+1);
  memcpy(lfs_path, path, strlen(path)+1);

  //checking the parent inode's datapointers
  int i;
  for (i=0; i<NUMBER_OF_DATAPOINTERS; i++) {
    if(lfs_inode->datapointer[i] != -1) {
      //set the path back to the inode we're searching for
      memset(lfs_path, 0, strlen(path)+1);
      memcpy(lfs_path, path, strlen(path)+1);

      if(!strcmp(basename(lfs_path), ((inode *) (lfs_disk_in_memory +
(lfs_inodeArray[lfs_inode->datapointer[i]]) * BLOCK_SIZE))->name)) {
        res = ((inode*) (lfs_disk_in_memory + (lfs_inodeArray[lfs_inode-
>datapointer[i]] * BLOCK_SIZE)))->ID;
        free(lfs_inode);
        free(lfs_path);

        //return the inode
        return res;
      }
    }
  }
  //checking the parent inode's indirectDataPointers
  if (lfs_inode->indirectDataPointer != -1) {
    int *lfs_indirectPointersArray;
    lfs_indirectPointersArray = (int *)(lfs_disk_in_memory + (lfs_inode-
>indirectDataPointer * BLOCK_SIZE));

    int j;
    for(j=0; j<NUMBER_OF_INDIRECTPOINTERS; j++) {
      if(lfs_indirectPointersArray[i] != -1) {
```

```
        memset(lfs_path, 0, strlen(path)+1);
        memcpy(lfs_path, path, strlen(path)+1);
        if(!strcmp(basename(lfs_path),((inode *)(lfs_disk_in_memory +
(lfs_inodeArray[lfs_indirectPointersArray[i]])*BLOCK_SIZE))->name)) {
          res = ((inode *)(lfs_disk_in_memory +
(lfs_inodeArray[lfs_indirectPointersArray[i]]*BLOCK_SIZE)))->ID;

          free(lfs_inode);
          free(lfs_path);
          return res;
        }
      }
    }
  }

  //no inode found
  free(lfs_inode);
  free(lfs_path);
  return -1;
}

//CREATE INODE METHOD

int lfs_createInode(char * path, int type) {
  printf("createInode method called\n");

  //create inode pointer
  inode *lfs_inode;
  lfs_inode = malloc(sizeof(inode));
  memset(lfs_inode, 0, sizeof(inode));

  //copy the path
  char *lfs_path;
  lfs_path = malloc(strlen(path)+1);
  memset(lfs_path, 0, strlen(path)+1);
  memcpy(lfs_path, path, strlen(path)+1);

  //set the inode type
  lfs_inode->type = type;

  //set the inode name
  if(strlen(basename(lfs_path)) < MAX_LENGTH) {
    memset(lfs_inode->name, 0, strlen(basename(lfs_path)));
    memcpy(lfs_inode->name, basename(lfs_path), strlen(basename(lfs_path)));
  }
  else {
    memset(lfs_inode->name, 0, strlen(basename(lfs_path)));
    memcpy(lfs_inode->name, basename(lfs_path), MAX_LENGTH);
  }

  //set the modify and access stamps
  lfs_inode->modify = time(NULL);
  lfs_inode->access = time(NULL);

  //set the size
  lfs_inode->size = 0;
```

```
  int j;
  for(j=0; j<NUMBER_OF_DATAPOINTERS; j++) {
    lfs_inode->datapointer[j] = -1;
  }

  lfs_inode->indirectDataPointer = -1;

  //find a free space in the array of inodes
  int i;
  for(i=0; i<NUMBER_OF_INODES; i++) {
    if(lfs_inodeArray[i] == -1) {
      lfs_inode->ID = i;
      //insert the new inode into the array
      lfs_inodeArray[i] = lfs_insertData((char *) lfs_inode, sizeof(inode));

      //set the path back to the inode we're creating
      memset(lfs_path, 0, strlen(path)+1);
      memcpy(lfs_path, path, strlen(path)+1);

      //get the parent inode
      int lfs_inodeID;
      lfs_inodeID = lfs_findInodeID(dirname(lfs_path));

      memset(lfs_inode, 0, sizeof(inode));
      memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[lfs_inodeID] *
BLOCK_SIZE), sizeof(inode));

      //set the parent datapointer
      int j;
      for(j=0; j<NUMBER_OF_DATAPOINTERS; j++) {
        if(lfs_inode->datapointer[j] == -1) {
          lfs_inode->datapointer[j] = i;

          //reinsert the parent with the new inode set to its datapointer
          lfs_inodeArray[lfs_inodeID] = lfs_insertData((char *) lfs_inode,
sizeof(inode));

          free(lfs_inode);
          free(lfs_path);

          return 0;
        }
      }

      //set the parent indirectDataPointer
      if(lfs_inode->indirectDataPointer == -1) {
        //no indirectDataPointers, need to create array
        int *lfs_arrayIndirectPointers;
        lfs_arrayIndirectPointers = malloc(NUMBER_OF_INDIRECTPOINTERS *
sizeof(int));
        int l;
        for(l=0; l<NUMBER_OF_INDIRECTPOINTERS; l++) {
          lfs_arrayIndirectPointers[l] = -1;
        }
        //set empty array to inode
        lfs_inode->indirectDataPointer = lfs_insertData((char *)
lfs_arrayIndirectPointers, NUMBER_OF_INDIRECTPOINTERS * sizeof(int));
```

```
        free(lfs_arrayIndirectPointers);
      }
      //get the parent indirectDataPointers array
      int *lfs_indirectPointersArray;
      lfs_indirectPointersArray = lfs_disk_in_memory + (lfs_inode-
>indirectDataPointer * BLOCK_SIZE);

      int m;
      for(m = 0; m<NUMBER_OF_INDIRECTPOINTERS; m++) {
        if (lfs_indirectPointersArray[m] == -1) {
          lfs_indirectPointersArray[m] = i;
          //set the updated indirectDataPointers array
          lfs_inode->indirectDataPointer = lfs_insertData((char *)
lfs_indirectPointersArray, NUMBER_OF_INDIRECTPOINTERS * sizeof(int));
          //insert the updated parent inode back into the array
          lfs_inodeArray[lfs_inodeID] = lfs_insertData((char *) lfs_inode,
sizeof(inode));

          free(lfs_indirectPointersArray);
          free(lfs_inode);
          free(lfs_path);

          return 0;
        }
      }
    } /*No space for a new inode*/ else {
      if (i == NUMBER_OF_INODES-1) {
        free(lfs_inode);
        free(lfs_path);

        return -ENOMEM;
      }
    }
  }

  free(lfs_inode);
  free(lfs_path);

  return 0;
}

//REMOVE INODE METHOD

int lfs_removeInode(char * path) {
  printf("remove inode method called\n");

  //find the inode
  int lfs_inodeID;
  lfs_inodeID = lfs_findInodeID(path);

  //find the parent inode
  int lfs_parentInodeID;
  lfs_parentInodeID = lfs_findInodeID(dirname(path));

  //create an inode pointer
  inode *lfs_inode;
```

```
  lfs_inode = malloc(sizeof(inode));

  //get the parent inode
  memset(lfs_inode, 0, sizeof(inode));
  memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[lfs_parentInodeID] *
BLOCK_SIZE), sizeof(inode));

  //remove the inode datapointer from the parent inode
  int i;
  for (i=0; i<NUMBER_OF_DATAPOINTERS; i++) {
    if(lfs_inode->datapointer[i] == lfs_inodeID) {
      lfs_inode->datapointer[i] = -1;
    }
  }

  //check indirectDataPointers
  if(lfs_inode->indirectDataPointer != -1) {
    int *lfs_indirectPointersArray;
    lfs_indirectPointersArray = lfs_disk_in_memory + (lfs_inode-
>indirectDataPointer * BLOCK_SIZE);

    int j;
    for(j=0; j<NUMBER_OF_INDIRECTPOINTERS; j++) {
      if(lfs_indirectPointersArray[j] == lfs_inodeID) {
        lfs_indirectPointersArray[j] = -1;
        lfs_inode->indirectDataPointer = lfs_insertData((char *)
lfs_indirectPointersArray, NUMBER_OF_INDIRECTPOINTERS * sizeof(int));
      }
    }
  }

  //insert the updated parent inode into array of inodes, and remove the
original inode from the array
  lfs_inodeArray[lfs_parentInodeID] = lfs_insertData((char *) lfs_inode,
sizeof(inode));
  lfs_inodeArray[lfs_inodeID] = -1;

  free(lfs_inode);

  return 0;
}

//INSERT DATA METHOD

int lfs_insertData(char * data, int size) {
  int block;
  printf("insertData method called\n");

  block = lfs_block;
  //insert the data into the disk in memory
  memset(lfs_disk_in_memory + (lfs_block * BLOCK_SIZE), 0, size);
  memcpy(lfs_disk_in_memory + (lfs_block * BLOCK_SIZE), data, size);

  //incriment block
  lfs_block++;

  if(lfs_block % (SEGMENT_SIZE / BLOCK_SIZE) == 0) {
```

```
    //have to begin a new segment, write the inode array at the beginning of
the segment
    memset(lfs_disk_in_memory + (lfs_segment * SEGMENT_SIZE), 0, 32 *
BLOCK_SIZE);
    memcpy(lfs_disk_in_memory + (lfs_segment * SEGMENT_SIZE), lfs_inodeArray,
32 * BLOCK_SIZE);

    //write segment to file
    lfs_write_segment(lfs_segment, ((char *)(lfs_disk_in_memory +
(lfs_segment * SEGMENT_SIZE))));

    //incriment block by 32 to save space for inode array
    lfs_block += 32;
    //incriment segment
    lfs_segment++;

    if(lfs_segment == NUMBER_OF_SEGMENTS){
      //file full, time to loop to beginning
      lfs_block = 32;
      lfs_segment = 0;
    }
    //clean the old inode array
    lfs_cleaner();

    return block;
  }
  return block;
}

//CLEANER METHOD
int lfs_cleaner(void) {
  printf("cleaner method called\n");

  //create an inode pointer
  inode *lfs_inode;
  lfs_inode = malloc(sizeof(inode));

  //create an empty new array of inodes
  int *lfs_newinodeArray;
  lfs_newinodeArray = malloc(NUMBER_OF_INODES * sizeof(inode));

  //initialize new array of inodes
  int m;
  for(m=0; m<NUMBER_OF_INODES; m++) {
    lfs_inodeArray[m]=-1;
  }

  //check each inode in the old inode array
  int i;
  for (i=0; i<NUMBER_OF_INODES; i++) {
    if(lfs_inodeArray[i] != -1) {
      //found an inode. Get the inode
      memset(lfs_inode, 0, sizeof(inode));
      memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[i] *
BLOCK_SIZE), sizeof(inode));

      if(lfs_inode->type == 0) {
```

```
        //the inode is a directory, copy it over.
        lfs_newinodeArray[i] = lfs_insertData((char *) lfs_inode,
sizeof(inode));
      } else {
        //The inode is a file, check its datapointers.
        int j;
        for(j=0; j<NUMBER_OF_DATAPOINTERS; j++) {
          if(lfs_inode->datapointer[j] != -1) {
            //found data
            lfs_inode->datapointer[j] = lfs_insertData(lfs_disk_in_memory +
(lfs_inode->datapointer[j] * BLOCK_SIZE), BLOCK_SIZE);
          }
        }
        //check indirectDataPointers
        if(lfs_inode->indirectDataPointer != -1) {
          //get the indirectDataPointers array
          int *lfs_indirectPointersArray;
          lfs_indirectPointersArray = lfs_disk_in_memory + (lfs_inode-
>indirectDataPointer * BLOCK_SIZE);

          int k;
          for(k = 0; k<NUMBER_OF_INDIRECTPOINTERS; k++) {
            if (lfs_indirectPointersArray[k] != -1) {
              //found data
              lfs_indirectPointersArray[k] =
lfs_insertData(lfs_disk_in_memory + (lfs_indirectPointersArray[k] *
BLOCK_SIZE), BLOCK_SIZE);
            }

          }
        //set the updated indirectDataPointers array
        lfs_inode->indirectDataPointer = lfs_insertData((char *)
lfs_indirectPointersArray, NUMBER_OF_INDIRECTPOINTERS * sizeof(int));

        free(lfs_indirectPointersArray);
        }
        //insert the updated inode into the new inode array
        lfs_newinodeArray[i] = lfs_insertData((char *) lfs_inode,
sizeof(inode));
      }
    }
  }
  //overrwite the old inode array with the new updated inode array
  memset(lfs_inodeArray, 0, NUMBER_OF_INODES);
  memcpy(lfs_inodeArray, lfs_newinodeArray, NUMBER_OF_INODES);

  free(lfs_newinodeArray);
  free(lfs_inode);

  return 0;
}

//WRITE SEGMENT METHOD

int lfs_write_segment(int segment, const char * data) {
  printf("writeSegment method called\n");
  FILE *file;
```

```
  file = fopen(HARDDISK,"w+");
  if(file == NULL) {
    perror("Write Segment: Error opening harrdisk file\n");
    return 1;
  }

  char * lfs_segment_data;
  lfs_segment_data = malloc(SEGMENT_SIZE);

  memset(lfs_segment_data, 0, SEGMENT_SIZE);
  memcpy(lfs_segment_data, data, SEGMENT_SIZE);

  if(fseek(file, (segment * SEGMENT_SIZE), SEEK_SET) != 0) {
    perror("Write Segment: Could not set stream position in harddisk");
    return 1;
  }
  fputs(lfs_segment_data, file);
  fclose(file);
  return 0;
}

//UTIME METHOD

int lfs_utime(const char * path, struct utimbuf * utime) {
  printf("utime method called\n");

  //create an inode pointer
  inode *lfs_inode;
  lfs_inode = malloc(sizeof(inode));

  //find the inode
  int lfs_inodeID;
  lfs_inodeID = lfs_findInodeID(path);

  //get the inode
  memset(lfs_inode, 0, sizeof(inode));
  memcpy(lfs_inode, lfs_disk_in_memory + (lfs_inodeArray[lfs_inodeID] *
BLOCK_SIZE), sizeof(inode));

  //set the access time
  if(utime != NULL) {
    lfs_inode->modify = time(NULL);
    lfs_inode->access = utime->actime;
  } else {
    lfs_inode->modify = time(NULL);
    lfs_inode->access = time(NULL);
  }

  //insert the updated inode into the array of inodes
  lfs_inodeArray[lfs_inodeID] = lfs_insertData((char *) lfs_inode,
sizeof(inode));

  free(lfs_inode);

  return 0;
}
```

```
int main( int argc, char *argv[] ) {
  lfs_init();
      fuse_main( argc, argv, &lfs_oper );

      return 0;
}
```

7.2 Appendix II: Source code for lfsTest

```
#!/bin/bash

cd /tmp/lfs-mountpoint2

echo Test 1: Will now attempt to create a file, and write to it.

sleep 1
touch newfile
echo "something" > newfile

echo The text written to newfile is:
cat newfile

sleep 1
echo Will now create a new directory.
sleep 1
mkdir newdir

echo The root now contains the following:
ls
sleep 1

echo will now enter 'newdir' and attempt to create a file

sleep 1
cd newdir

touch anotherfile

echo 'newdir' now contains the following:
ls
```

7.3 Appendix II: Source code for lfsTest2

```
#!/bin/bash

cd /tmp/lfs-mountpoint2

echo Test 2: Will now attempt to create a file and directory, and then delete
the file and directory
sleep 1
touch file_for_deleting

mkdir directory_for_deleting

echo The root folder now contains the following:
```

```
ls
sleep 1

echo Will now attempt to delete the contents of root

rm file_for_deleting
rm -r directory_for_deleting

echo The root folder now contains the following:
ls
```

7.4 Appendix III: Source code for lfsTest3

```
#!/bin/bash

cd /tmp/lfs-mountpoint2

echo Test 3: Attempting to create a file with a name that is too long

sleep 1

touch
filewithanamethatstoolong12345678910111213141516171819202122232425262728293031323

echo The root folder now contains the following:
ls
```

7.5 Appendix IV: Source code for lfsTest4

```
#!/bin/bash

cd /tmp/lfs-mountpoint2
echo Test 4: Will now attempt to create 5 files with a large amount of text

sleep 1

for i in {0..5}
do

      echo
hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohe
llohellohellohellohellohellohellohellohellohellohellohellohellohellohellohell
ohellohellohellohellohellohellohellohellohellohellohellohellohellohellohelloh
ellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohel
lohellohellohellohellohellohellohellohellohellohellohellohellohellohellohello
hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohe
llohellohellohellohellohellohellohellohellohellohellohellohellohellohellohell
ohellohellohellohellohellohellohellohellohellohellohellohellohellohellohelloh
ellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohel
lohellohellohellohellohellohellohellohellohellohellohellohellohellohellohello
hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohe
llohellohellohellohellohellohellohellohellohellohellohellohellohellohellohell
```

```
ohellohellohellohellohellohellohellohellohellohellohellohellohellohellohelloh
ellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohel
lohellohellohellohellohellohellohellohellohellohellohellohellohellohellohello
hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohe
llohellohellohellohellohellohellohellohellohellohellohellohellohellohellohell
ohellohellohellohellohellohellohellohellohellohellohellohellohellohellohelloh
ellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohel
lohellohellohellohellohellohellohellohellohellohellohellohellohellohellohello
hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohe
llohellohellohellohellohellohellohellohellohellohellohellohellohellohellohell
ohellohellohellohellohellohellohellohellohellohellohellohellohellohellohelloh
ellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohel
lohellohellohellohellohellohellohellohellohellohellohellohellohellohellohello
hellohellohellohellohellohellohellohellohellohellohellohellohellohellohello >
"File${i}.txt"
echo the root folder now contains the following:
ls
      sleep 1
done

echo The files each contain the following text:
cat File0.txt
```

7.6 Appendix V: Source code for lfsTest5

```
#!/bin/bash

cd /tmp/lfs-mountpoint2

echo will now attempt to create a file with 1MB of data
sleep 1

dd if=/dev/zero of=file.txt count=1024 bs=1024

echo The root folder now contains the following:
ls
```