dm510 project 3

Danny Jensen
danje14

Group:
Lea Fog-Fredsgaard
Michelle Dung Hoang

April 2018

Contents

1 Introduction

In this project a file system must be implemented. The user of the file system must be able to create and navigate through the newly created folders.
The user should also be able to create files that the user can then write and read from using *echo* and *cat*.
All these folders and file should be stored in an inode.

2 Design

The first design choices was how the inode structure should look. Since it was a requirement that an access time and a modified time was present this was the first thing to look at. The the choice was made that the inode also had to have its path as its name so that it would be easier to search for a given inode. every inode whould then also have a "uniqe" number to help locate the place where it would be saved onto the disk.
Sadly the indirect pointers was not implemented because of errors. But the inode holds pointers to all its children which makes it easy to navigate through every layer.
The inode of-cause also need some place to hold data, this was first tried with and string array but the idea was dropped because it needed a fixed length and that also meant that the inode would use more space than necessary. Instead a void pointer is used that points to the place in memory the data is put and an integer for the length is added.

Since the structure must be inside blocks that must be inside segments. A block structure is needed, this block should hold the inode and some integer so as to identify if the inode is usable or not.
Since it was impossible to get the segment part working it was dropped and the blocks are simply saved to the disk.
To retrieve the inode, the first plan was to let a function run through the linked list structure but that was dismissed as it would be much harder than simply have an array with every inode in that is in the memory.
To save everything in a file, the first idea was to save every change to a file as soon as it was made,but this would make the process run slower because of saves and retrieves all the time, that is why a simple void variable is chosen. This variable is updated with all new info and should be retrieved when the process starts and saved to a file when the process is shut down, sadly there was not enough time to implement this part.

Another design decision was that it would not be possible to delete a folder with contents in it, this design decision was taken because the *unlink* function should not be implemented and that meant that there was no way of deleting a file.

3 Implementation

As mentioned in the Design chapter, the inode has to have ways to store the time of last access and modified, these variables are saved as *time_t*. The inode also needs a type to tell if it is a folder or a file, for this a simple int variable is used, if the variable is 0 it is a folder and if it is 1 the inode is a file. The inode_direct variable is a set of pointers to all the direct children of the inode, and the inode_path is the inode's name and its path as a string.
The *current_inode_amount* variable is the one that holds the number of variables. No two inodes should have the same number. This variable is incremented each time a new inode is created.

The *lfs_init* function simply initializes all the variables such as *disk* and *array_inode* and allocate space for it in memory. *lfs_init* also creates the root inode that is the root folder, and puts it into *array_inode* at place 0.

There are three diffrent functions to find inodes, *find_inode*, *find_inode_dir* and *find_inode_file* they pretty much do the same. They look through *array_inode* to find the inode with the given path and returns it. The difference is that *find_inode* does not care what type of inode it is, either folder or file, where *find_inode_dir* finds only a folder with that path and *find_inode_file* only finds the file with that path.

```
1 if ((strcmp(array_inode[i] -> node -> inode_path, path) == 0) && (
      array_inode[i] -> node -> inode_type == 0)){
```

These three functions could have been easily put into one function but the time did not allow for optimization of the code.

To find the parent of a file or folder a simple function named **find_inode_parent* was created. It copes the path to another string variable, this is done because *path* is a *const char* type and can not a should not be changed. on This new copy the *dirname* function is used, it is a smart function that removes everything after the last / in a string. This new string is then the parent path and can simply be put into a *find_inode* function to return the parent of a path.

```
1 inode *find_inode_parent(const char *path){
2   char *path_copy;
3   path_copy = strdup(path);
4   return find_inode(dirname(path_copy));
5 }
```

lfs_getattr simply copies the data from the inode, found by the path, over into the *stbuf* with the data that was given. Just to ensure that nothing is wrong *res* is set to *ENOENT* if there is no inode with that path or that something want wrong and the type cant be right.

lfs_readdir first creates a inode to put the hopefully found inode into, with the given path. To find this inode *find_inode_dir* is used because as the name states it needs to return a folder. When the right inode is found, every path

of every child is then given to the *filler* function to return to the user. This is easily done because the inode has pointers to all its direct children, and an int variable that holds the number of children. Just like *lfs_getattr*, *lfs_readdir* will also return *ENOENT* if there is no inode with that path.

```
for(int i = 0; i < inod -> inode_sub_num; i++){
  filler(buf, basename(((inode*)inod -> inode_direct[i]) ->
    inode_path), NULL, 0);
}
```

To simplify *lfs_mknod* and *lfs_mkdir* they where simply put into one function named *lfs_mk* that takes the path for the new inode and the type in form of an int

```
int lfs_mkdir(const char *path, mode_t mode) {
  int ret = lfs_mk(path, 0);
  return ret;
}
```

The *lfs_mknod* and *lfs_mkdir* could easily have been one liners there is no real meaning in creating an int variable to return, the return value from *lfs_mk* could just have been returned, but at the time this gave sense and as stated before sadly there was not enough time to optimize the code.
lfs_mk first finds the parent of the inode that is to be created, this is done by using *find_inode_parent*. then *lfs_mk* starts to allocate space for the new inode. There is also allocated space for *inode_data* which shall hold the data if it is a file, this should actually have been inside an if statement so as to not use unnecessary ram fr a folder inode which has no use for it. The size is set to 20 times the size of a void pointer, this was done to test the ability to save strings inside files, and should have been changed, but it was down prioritized over getting other things working.
Now that the new inode with the given path is created, the parent inode needs to get a pointer in its *inode_direct* and of cause its *inode_sub_num* needs to be incremented, here a wierd thing happened if a simple

```
parent -> inode_sub_num++;
```

was used first but then the the process froze, so

```
parent -> inode_sub_num = parent -> inode_sub_num+1;
```

was used and it fised the error.

Now *lfs_rmdir* was a bit tricky. First the parent is found for later use. Then the actual inode that is to be deleted is found, and since it needs to be a folder is *find_inode_dir* used.
If the folder has no children then it can be deleted, as stated in the design chapter this is so because if there was a file inside the folder to be deleted it should also be deleted but there is no way of deleting a file in the code.
If the folder can be deleted, the pointer in the parent is first deleted. This is done by setting the pointer to *NULL* and the simple run through every other pointer and placing it in the previous place in the array.

```
for(int j = i; j < parent -> inode_sub_num-1; j++){
  parent -> inode_direct[j] = parent -> inode_direct[j+1];
}
```

After this the last pointer is set to *NULL* and *inode_sub_num* of the parent is reduced by one.
The next part has some errors in it.

```
for(int i = 0; i < current_inode_amount; i++){
  if(strcmp(array_inode[i] -> node -> inode_path, inod ->
    inode_path) == 0){
    array_inode[i] = NULL;
    for(int j = i; j < current_inode_amount; j++){
      array_inode[j] = array_inode[j+1];
    }
    array_inode[current_inode_amount] = NULL;
    current_inode_amount--;
    break;
  }
}
```

In an effort to try and clean the data a *cleanup* function was meant to be added and this is the remnents of this function that was fogotten in the code. If *current_inode_amount* is reduced and then a new node is added the data could be stored over an inode that is still in use inside the disk. This will lead to nodes being lost and that is not a good thing. The reduction on the *array_inode* is okay because the "uniqe" number of the inodes are stored in *inode_number* of each inode.
This is a bit confusing so lets take a small example, every node is a folder:
3 nodes are created with *inode_number*, 1,2 and 3. Now *current_inode_amount* is 3. node 2 is then deleted and *current_inode_amount* is reduced. Then a new node is created this means that it get 3 in its *inode_number*, this means we have 1, 3 and 3. And since the *inode_number* is used to get the pointer to where the inode is stored this means that both inodes are stored the same place which is not possible. This means that the first inode with *inode_number* that is 3 is overwritten.

lfs_read is pretty simple, it just finds the inode with the help of *find_inode_file* and copies everything inside the *inode_data* to the buffer and returns the number of bytes stored in the *inode_size* of the inode.

lfs_write finds the inode using *find_inode_file* copies the string over into *inode_data*. It then finds the length of the string and puts it into the *inode_size* of the inode.

```
memcpy(inod -> inode_data, buf, strlen(buf));
inod -> inode_size = strlen(buf);
```

When is done with putting the string into the inode, the inode is then put into the disk with the new information. The inode is then put into a block and*array_inode* is updated. This part gave some problems because it did not make so much sense why *array_inode* should be updated, it only has pointers

to the inodes inside it. But an error was returned every time the inode was read.

lfs_utime simply retrieves the inode and updates the *inode_access* and *inode_modified* time with the time given from the *utb*.

4 Test

The test was video captured so it will be refereed to with time stamps here instead of showing pictures.

First the file system is mounted, this happens at 00.05. And then go into the newly mounted file system.
An *ls* command is run to show that the folder is empty at 00.19. And right after that a folder is created named *dir1*
At 00.29 another *ls* is run to show that the *mkdir* and the *ls* functions actually work, and as shown a directory was created with the name *dir1*.
Next a file is created with the name file1 using the *touch* command, this happens at 00.34
Just to show that *file1* was created a *ls* command was run again. And as it shows *file1* was created.
at 00.46 the *cd* command was checked by going into the directory by the name *dir1*. And as the path shows, *dir1* was successfully entered. Just to show that this directory is also empty a *ls* command is run in this directory too.
Now to navigate back to the parent folder of *dir1* the *cd ..* is used, this happens at 0.52. And this is also successful as shown by the path to the right.
A new *ls* command is used to show that the folder and file both still are present in the root directory, this happens at 0.65.
A new directory is made using *mkdir* by the name *dir2* and then *cd* is used to navigate into this folder, at 1.00. Then at 1.17 another file is created using the *touch* command, this new file is named *file2*
at 1.44 *file2* is written to using the *echo* command, the string that was written was *"hello from file2"*.
Now to see if the string added to *file2* really was added and saved, a *cat* command is used on *file2* at 1.50. And yes the string was added and saved as it is printed out to the terminal.
Now go back to the root and try to delete folders. To delete a folder the *rmdir* command is used, and the target directorty to remove is *dir1* this happens at 2.06. a *ls* command is run to show that *dir1* was removed from the root directory.
dir1 was empty when removed so what happens if the directory that is to be deleted not empty is. This test is performed at 2.15. *dir2* is not empty as it holds *file2* but the *rmdir* command is still performed on *dir2*. As shown in the output the folder *dir2* is not empty and *rmdir* failed to remove it.
Now to test the *ls* function a bit *ls* is performed with *dri2* as a path while standing in the root directory, this happens at 2.33. As shown *dir2* holds *file2*,

so this also works like planned.
Next to test if *cat* can do the same. At 2.43 *cat* is given the path *dir2/file2* while also still in the root directory. *"hello from file2"* was printed out to the terminal and this was also a huge success.

5 Conclusion

The file system is able to create and it is possible to navigate through the directories. It is also possible to create, write to and read from a file.
Sadly it is not possible to shut the file system down and start it back up again without loosing all the data, but this should not be that hard to do since everything is in a *disk* variable.
The segmentation part is not completed either but if the user does not use to long strings this is no problem. And the code could use some cleanup from other things that was created in an attempt to fix other errors.
But all in all the code runs and it is possible to use it as a rudimentary file system.

6 Appendix

6.1 lfs.c

```
#include <fuse.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <fcntl.h>
#include <libgen.h>
#include <sys/types.h>
#include <unistd.h>

#define SEGMENTS 4      /* segments containing the blocks, and array_inode */
#define BLOCK 4         /* Blocks containing the data */
#define BLOCK_SIZE  256
#define SEGMENT_SIZE 1024


typedef struct inode{
  ino_t inode_number;       /* id of the node */
  char *inode_path;         /* path to where the file or directory is placed on the computer*/
  int inode_type;           /* 0 directory, 1 file */
  size_t inode_sub_num;     /* size of sub inodes in current inode */
  time_t inode_access;      /* time of last access */
  time_t inode_modified;    /* time of last modification */
  void **inode_direct;      /* pointer to all sub files or directories */
  //int **inode_indirect;   /* pointer to pointer to datablocks */
  size_t inode_size;        /* size of data in files */
  void *inode_data;         /* data */

} inode;

typedef struct block {
  struct inode *node;
  int in_use;
} block;



inode *find_inode(const char* path);
inode *find_inode_dir(const char* path);
inode *find_inode_file(const char* path);
inode *find_inode_parent(const char* path);
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_mk(const char *path, int type);
int lfs_mknod(const char* path, mode_t mode, dev_t rdev);
int lfs_mkdir(const char *path, mode_t mode);
int lfs_rmdir(const char *path);
```

```
49 int lfs_open(const char *, struct fuse_file_info *);
50 int lfs_read(const char *, char *, size_t, off_t, struct
       fuse_file_info *);
51 int lfs_release(const char *path, struct fuse_file_info *fi);
52 int lfs_write(const char *path, const char *buf, size_t size, off_t
        offset, struct fuse_file_info *fi );
53 int lfs_utime (const char *path, struct utimbuf *utb);
54
55
56
57
58 block **array_inode;                /* pointer to the inodes */
59 int current_inode_amount = 0;       /* current amount of inodes,
       increased when a new node is made */
60 void *disk;                         /* disk containing the segments
        */
61 int segment;
62 //int number_of_inodes = 0;
63
64 int lfs_init(void){
65   printf("---init---\n");
66   inode *inod;
67   inod = malloc(sizeof(inode));   /* allocate space for the inode
       */
68   if(inod == NULL){
69     return -ENOMEM;
70   }
71   inod -> inode_path = calloc(200,sizeof(char));  /* allocate space
       for the inode path */
72   if(inod -> inode_path == NULL){
73     return -ENOMEM;
74   }
75   char *root_path = "/";
76   /* initializing the rootnode */
77   inod -> inode_number = 0;           /* Root node */
78   strcpy(inod -> inode_path, root_path); /* initialize the root
       path to "/" */
79   inod -> inode_type = 0;
80   inod -> inode_sub_num = 0;
81   inod -> inode_size = 0;
82   inod -> inode_access = time(NULL);
83   inod -> inode_modified = time(NULL);
84   inod -> inode_direct = malloc(15*sizeof(inode));
85   //printf("hello1\n");
86   array_inode = malloc(1024 * sizeof(inode));
87   if(array_inode == NULL){
88     free(inod);
89     return -ENOMEM;
90   }
91
92   disk = malloc(SEGMENTS * SEGMENT_SIZE);
93   if(disk == NULL){
94     free(inod);
95     free(array_inode);
96     return -ENOMEM;
97   }
98 /*
```

```
99    segment = malloc(BLOCK * BLOCK_SIZE);
100   if(segment == NULL){
101     free(inod);
102     free(array_inode);
103     free(disk);
104     return -ENOMEM;
105   }
106 */
107   //printf("hello2\n");
108   block *bl;
109   bl = malloc(sizeof(block));
110
111   if(bl == NULL){
112     free(inod);
113     free(array_inode);
114     free(disk);
115     //free(segment);
116     return -ENOMEM;
117   }
118
119   bl -> node = inod;
120   bl -> in_use = 1;
121
122   array_inode[0] = bl; /* places the rootnode in the 0'th place */
123   current_inode_amount++;
124   memcpy((disk + (current_inode_amount-1 * BLOCK_SIZE)), inod,
        sizeof(inode)); /* copies from inod to the next available space
         on the disc  */
125
126   //printf("GOODBYE from init\n");
127   return 0;
128 }
129
130 /* find inodes */
131 inode *find_inode(const char* path){
132   printf("---find_inode---\n");
133   for(int i = 0 ; i < current_inode_amount ; i++) {
134     printf("      find_inode: %i out of %i iterations and path: %s\
        n", i, current_inode_amount, path);
135
136     if (strcmp(array_inode[i] -> node -> inode_path, path) == 0){
        /* if it finds the path */
137       printf("      the path of the inode is found: %s\n",
        array_inode[i] -> node -> inode_path);
138       inode *inod;
139       //inod = malloc(sizeof(inode));
140       inod = array_inode[i] -> node;
141       return inod;                                  /* returns the
        inode*/
142     }
143   }
144   printf("no inode found \n");
145   return NULL;
146 }
147
148 /* find directory inodes */
149 inode *find_inode_dir(const char* path){
```

```
150   printf("---find_inode_dir---\n");
151   for(int i = 0 ; i < current_inode_amount ; i++) {
152     printf("      find_inode_dir: %i out of %i iterations and path
      %s\n", i, current_inode_amount, path);
153
154     /* if it finds the path and it is a directory */
155     if ((strcmp(array_inode[i] -> node -> inode_path, path) == 0)
      && (array_inode[i] -> node -> inode_type == 0)){
156       printf("      the path of the directory is found: %s\n",
      array_inode[i] -> node -> inode_path);
157       inode *inod;
158       inod = malloc(sizeof(inode));
159       inod = array_inode[i] -> node;
160       return inod;                                  /* returns the
      inode*/
161     }
162   }
163   printf("no find_inode_dir found \n");
164   return NULL;
165 }
166
167 /* find file inodes */
168 inode *find_inode_file(const char *path){
169   //inode *inod;
170   printf("---find_inode_file---\n");
171   for(int i = 0 ; i < current_inode_amount ; i++) {
172     printf("      find_inode_file: %i out of %i iterations and path
      %s\n", i, current_inode_amount, path);
173
174     /* if it finds the path and it is a file */
175     if ((strcmp(array_inode[i] -> node -> inode_path, path) == 0)
      && (array_inode[i] -> node -> inode_type == 1)){
176       printf("      the path of the file is found: %s\n",
      array_inode[i] -> node -> inode_path);
177       return (array_inode[i] -> node);
          /* returns the inode*/
178     }
179   }
180   printf("no find_inode_file found \n");
181   return NULL;
182 }
183 /* find parent inode */
184 inode *find_inode_parent(const char *path){
185   printf("---find_parent---\n");
186   char *path_copy;
187   path_copy = strdup(path);
188   return find_inode(dirname(path_copy));     /* copies the path up
      to, but not including, the final '/' */
189 }
190
191
192
193
194 /*  */
195 int lfs_getattr( const char *path, struct stat *stbuf ) {
196   printf("---getattr---(path:  %s)\n", path);
197   int res = 0;
```

```
198   //printf("getattr: (path=%s)\n", path);
199
200   memset(stbuf, 0, sizeof(struct stat)); /* initialize stbuf with 0
        */
201   inode *inod;
202   //if(inod == NULL){
203   //  return -ENOENT;                    /* no such file og
        directory */
204   //}
205   //printf("getattr3\n");
206   inod = find_inode(path);      /* finds the inode */
207   if(inod == NULL){
208     return -ENOENT;
209   }
210   stbuf -> st_ino = current_inode_amount;
211   stbuf -> st_size = inod->inode_size;
212   stbuf -> st_atime = time(NULL);
213   stbuf -> st_mtime = inod->inode_modified;
214   stbuf -> st_ctime = inod->inode_access;
215   if( inod->inode_type == 0){       /* directory */
216     stbuf -> st_mode = S_IFDIR | 0755;
217     stbuf -> st_nlink = 2;
218   }else if(inod->inode_type == 1){    /* file */
219     stbuf -> st_mode = S_IFREG | 0777;
220     stbuf -> st_nlink = 1;
221   }else                               /* file */
222     res = -ENOENT;                    /* no such file og directory
        */
223   //printf("getattr end\n");
224   printf("      last access time: %li \n", inod->inode_access);
225   printf("      last modified time: %li\n\n\n", inod->
        inode_modified);
226   return res;
227 }
228
229
230 /* display all content of a directory */
231 int lfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler
        , off_t offset, struct fuse_file_info *fi ) {
232   printf("---readdir---\n");
233   (void) offset;
234   (void) fi;
235
236   inode *inod = malloc(sizeof(inode));
237   if(inod == NULL){
238     return -ENOENT;                 /* no such file og directory */
239   }
240
241   filler(buf, ".", NULL, 0);
242   filler(buf, "..", NULL, 0);
243   inod = find_inode_dir(path);      /* Finds the path given to
        lfs_readdir, in the current directory  */
244   /*
245   if(strcmp(path, "/") != 0){
246     return -ENOENT;
247   }*/
248
```

```
249   for(int i = 0; i < inod -> inode_sub_num; i++){
250     filler(buf, basename(((inode*)inod -> inode_direct[i]) ->
        inode_path), NULL, 0); /* Filler displays the content of the
        buffer with all inodes in the current directory */
251   }
252   printf("---readdir end---\n");
253   //free(inod);
254   return 0;
255 }
256
257
258 int lfs_mk(const char *path, int type) {
259   printf("---mk---\n");
260   inode *parent;
261   parent = malloc(sizeof(inode));
262   if(parent == NULL){
263     return -ENOMEM;
264   }
265   printf("mk2\n");
266   parent = find_inode_parent(path);
267
268   inode *inod;                    /* file or directory */
269   inod = malloc(sizeof(inode));   /* allocate space for the inode
        */
270   if(inod == NULL){
271     return -ENOMEM;
272   }
273   printf("mk3\n");
274   inod -> inode_path = malloc(sizeof(char));  /* allocate space for
        the inode path */
275   if(inod ->inode_path == NULL){
276     return -ENOMEM;
277   }
278   printf("mk4\n");
279   inod -> inode_number = current_inode_amount;          /* current
        node */
280   strcpy(inod->inode_path, path);                      /* initialize
        the path */
281   inod -> inode_type = type;
282   inod -> inode_sub_num = 0;
283   inod -> inode_size = 0;
284   inod -> inode_access = time(NULL);
285   inod -> inode_modified = time(NULL);
286   inod -> inode_direct = malloc(15*sizeof(inode));
287   inod -> inode_data = malloc(20*sizeof(void*));
288   block *bl;
289   bl = malloc(sizeof(block));
290   if(bl == NULL){
291     return -ENOMEM;
292   }
293   bl -> node = inod;
294   bl -> in_use = 1;
295   array_inode[current_inode_amount] = bl; /* places the current
        block with the inode in the array  */
296   current_inode_amount++;
297   memcpy((disk + (current_inode_amount * BLOCK_SIZE)), inod, sizeof
        (inode)); /* copies from inode to the next available space on
```

```
    the disc  */
298   parent -> inode_direct[parent -> inode_sub_num] = inod; /* points
        to the new inode */
299   parent -> inode_sub_num = parent -> inode_sub_num+1;        /*
        increment size of the inode */
300   return 0;
301 }
302 /* uses lfs_mk to create a file */
303 int lfs_mknod(const char* path, mode_t mode, dev_t rdev) {
304   printf("---mknod---\n");
305   int ret = lfs_mk(path, 1);
306   return ret;
307 }
308
309 /* uses lfs_mk to create a directory */
310 int lfs_mkdir(const char *path, mode_t mode) {
311   printf("---mkdir---\n");
312   int ret = lfs_mk(path, 0);
313   return ret;
314 }
315
316 int lfs_rmdir(const char *path) {
317   printf("---rmdir---\n");
318   inode *parent;
319   parent = malloc(sizeof(inode));
320   if(parent == NULL){
321     return -ENOMEM;
322   }
323   parent = find_inode_parent(path);
324
325   inode *inod;                      /* directory to be deleted */
326   inod = malloc(sizeof(inode));     /* allocate space for the inode
        */
327   if(inod == NULL){
328     return -ENOMEM;
329   }
330   inod = find_inode_dir(path);      /* the path to the directory to
        be deleted */
331
332   if(inod -> inode_sub_num == 0){          /* is there sub
        directories */
333     for(int i = 0; i < parent -> inode_sub_num; i++){
334       //inod = parent -> inode_direct[i]; /* finds the pointer to
        the parent directory */
335       if(strcmp(((inode*)parent -> inode_direct[i]) -> inode_path,
        inod -> inode_path) == 0){ /* finds the right path in the
        parent  */
336         parent -> inode_direct[i] = NULL;
337         for(int j = i; j < parent -> inode_sub_num-1; j++){
338           parent -> inode_direct[j] = parent -> inode_direct[j+1];
        /* when found, shift everything one place to the left */
339         }
340         parent -> inode_direct[parent -> inode_sub_num-1] = NULL;
341         parent -> inode_sub_num--;
342         break;
343       }
344     }
```

```
345
346     for(int i = 0; i < current_inode_amount; i++){
347       if(strcmp(array_inode[i] -> node -> inode_path, inod ->
    inode_path) == 0){ /* finds the right path in the in the inode
    array  */
348         array_inode[i] = NULL;
349         for(int j = i; j < current_inode_amount; j++){
350           array_inode[j] = array_inode[j+1];  /* when found, shift
    everything one place to the left */
351         }
352         array_inode[current_inode_amount] = NULL;
353         current_inode_amount--;
354         break;
355       }
356     }
357     return 0;
358   }
359
360   return -ENOTEMPTY;
361 }
362
363 //Permission
364 int lfs_open(const char *path, struct fuse_file_info *fi) {
365     printf("---Open--- path: %s\n", path);
366   return 0;
367 }
368
369 int lfs_read(const char *path, char *buf, size_t size, off_t offset
    , struct fuse_file_info *fi) {
370   printf("---Read--- path: %s\n", path);
371   inode *inod;
372   inod = find_inode_file(path);
373   memcpy(buf, inod -> inode_data, inod -> inode_size);
374   printf("---read end---\n" );
375   return inod -> inode_size;
376 }
377
378 int lfs_release(const char *path, struct fuse_file_info *fi) {
379   printf("---Release---  path: %s\n", path);
380   return 0;
381 }
382
383 int lfs_write(const char *path, const char *buf, size_t size, off_t
     offset, struct fuse_file_info *fi ) {
384   printf("---Write--- path: %s\n", path);
385   inode *inod;
386   inod = find_inode_file(path);
387   memcpy(inod -> inode_data, buf, strlen(buf));
388   inod -> inode_size = strlen(buf);
389   memcpy((disk + ((inod ->inode_number) * BLOCK_SIZE)), inod,
    sizeof(inode));
390   array_inode[inod ->inode_number] = NULL;
391   block *bl = malloc(sizeof(block));
392   bl -> node = inod;
393   bl -> in_use = 1;
394   array_inode[inod ->inode_number] = bl;
395
```

```
396   printf("---write end---\n" );
397   return size;
398 }
399
400 int lfs_utime (const char *path, struct utimbuf *utb) {
401   printf("---Utime---, path: %s\n", path);
402
403   inode *inod;
404   inod = find_inode(path);
405   inod -> inode_access = utb -> actime;
406   inod -> inode_modified = utb -> modtime;
407
408   return 0;
409 }
410
411 static struct fuse_operations lfs_oper = {
412   .getattr  = lfs_getattr,
413   .readdir  = lfs_readdir,
414   .mknod = lfs_mknod,
415   .mkdir = lfs_mkdir,
416   .unlink = NULL, //not needed
417   .rmdir = lfs_rmdir,
418   .truncate = NULL,
419   .open = lfs_open,
420   .read = lfs_read,
421   .release = lfs_release,
422   .write = lfs_write,
423   .rename = NULL, //not needed
424   .utime = lfs_utime
425 };
426
427 int main( int argc, char *argv[] ) {
428   lfs_init();
429   fuse_main( argc, argv, &lfs_oper );
430   return 0;
431 }
```

6.2 makefile

```
1  GCC = gcc
2  SOURCES = lfs.c
3  OBJS := $(patsubst %.c,%.o,$(SOURCES))
4  CFLAGS = -O2 -Wall -D_FILE_OFFSET_BITS=64 -DFUSE_USE_VERSION=25
5
6  .PHONY: lfs
7
8  ##
9  # Libs
10 ##
11 LIBS := fuse
12 LIBS := $(addprefix -l,$(LIBS))
13
14 all: lfs
15
16 %.o: %.c
17   $(GCC) $(CFLAGS) -c -o $@ $<
18
```

```
19 lfs: $(OBJS)
20   $(GCC) $(OBJS) $(LIBS) $(CFLAGS) -o lfs
21
22 clean:
23   rm -f $(OBJS) lfs
24
25 mount:
26   mkdir lfs-mountpoint1
27   chmod 755 lfs-mountpoint1/
28   ./lfs -f lfs-mountpoint1/
29
30 unmount:
31   fusermount -u lfs-mountpoint1/
32   rm -rf lfs-mountpoint1
```