DM510 - Operating Systems
Project 4: File System

Jeff Gyldenbrand (jegyl16)
Supervisor: Daniel Merkle
Southern University of Denmark

April 23, 2018

Contents

1 Introduction

This project is done in collaboration with Jonas Sørensen (joso216) and Simon D. Jørgensen (simjo16).

The goal of this project is to implement a file-system in linux using FUSE (file system in userspace, which is an interface for linux that let users create their own filesystem without touching kernel space. More specific, the implementation should include directories and files, and a way to create, delete and list them. Furthermore a way to open, close, read and write files. Lastly the implementation should handle a way to show size, access- and modification time-stamps, to files.

2 Design decisions

The file system is implemented with inodes and implemented such, that they have the behavior of a b-plus three. Where the inodes is the buckets of the three. A struct-type *inode_t* is developed, and has the elements *level*, *fill*, *keys* and *values*. The keys of the inode is highlighted in green, and their associated values is highlighted in grey. The inode is implented with size $n = 4$. See figure 1. If a node is a branch-node, it has n keys, $n + 1$ values. If a node is a leaf-node, is has n keys, n values. In figure 1, a leaf node is illustrated.

The filesystem is one big file denoted as filesystem.fs. Again, see figure 1, where filesystem.fs consist of "$0hello1world$". In this example, the zero is the first key in the root inode. This key is located in memory at adress a. The key 0 has the associated value *hello*, which is located in memory at adress b. The next key is 1 with associated value *world*, located at the memory adresses, respectively, c and d.

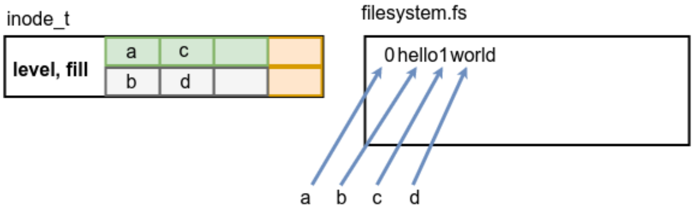

fig:1

In figure 2, an example of the case, where a key -1 with value *goodbye* is inserted, is illustrated. Here the pointers in the inode struct will be sorted and updated in acording to the rules of the b-plus tree, hence key -1 and its asociated value will be placed in the beginning of the inode.

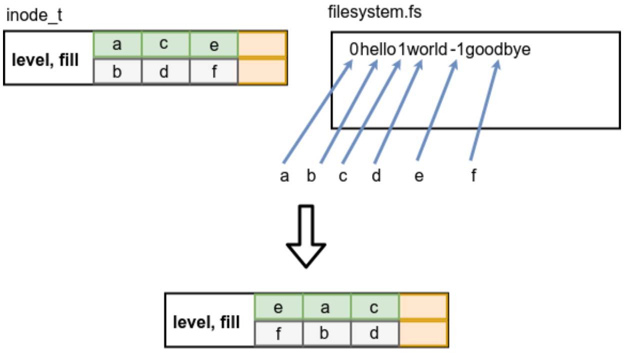

fig:2

To continue the example, a key 2 and its asociated value LOL is inserted. However, the inode is full. Therefore the node is splitted into an α and β inode, as seen in figure 3. A new root inode is created, with key c and associated pointers α and β, which point to, respectively, the two leaf nodes α and β. If a new key is to be inserted, and said key is less than c, said key would be assigned to the α node, otherwise it would be assigned to the β node. Notice the root is now a branch-node, thus having $n + 1$ values. This may seem odd, however, see it as this: c has a pointer to α and β. If another key was inserted, this would have a pointer to β and γ, and yet another key would have pointers to γ and δ.

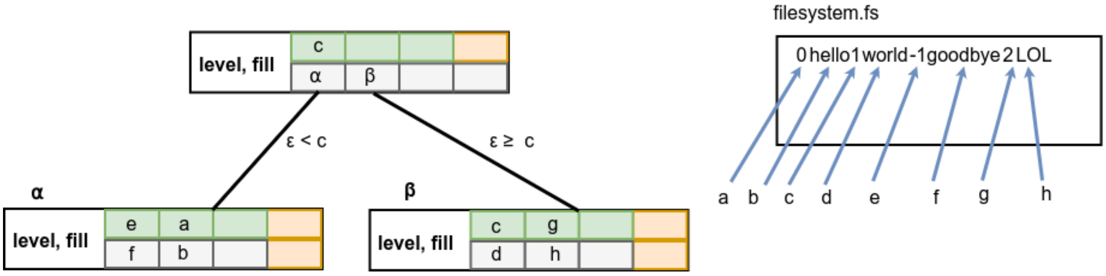

fig:3

Besides creating of new nodes, the implementation supports search for a given key, delete a key by replacing its pointer to zero, thus treating it as a NULL pointer, hence, now the spot is free. This is also how updates are handled; by deletion of a key, and insertion of a new one. All this will be elaborated in section 3.

3 Implementation

The examples in section 2 is just to illustrate the idea of the design. Here the keys are given as integers. The actual implementation of the b-plus three sorts lexographically, so the keys and values can be anything.

Several features have been developed to make the file system work. The datatructure for inode is implemented in inode.h and inode.c. The struct for inodes is shown in listing 1 below.

```
typedef struct{
  file_ptr file_ptr;
  size_t level;
  size_t fill;
  size_t keys[INODE_SIZE];
  size_t values[1 + INODE_SIZE];
  char data[];
} inode_t;
```

Listing 1: inode_t

Due to the amplitude of functions created to succesfully make a filesystem, it would be overwhelming to cover them all. Therefore only the inode implementation will be covered in details here.

The function *node_tree* is used to expand the b+ tree. This is done whenever a new root is created, and based on the new level, node tree will generate the same number of nodes. See figure 4, where the red nodes illustrates the newly expansion from root.

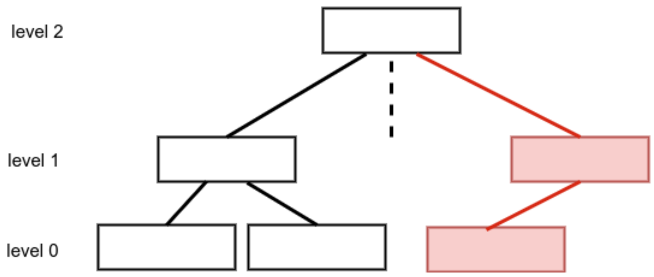

fig:4

```
inode_t * node_tree(FILE *fp, size_t level){
  inode_t n = {
    .file_ptr.size = sizeof(n) - sizeof(file_ptr),
    .level = level,
    .fill = level > 0
  };
  if(level){
     inode_t * result = node_tree(fp,level - 1);
     n.values[0] = result->file_ptr.pos;
     free(result);
  }
  return (inode_t *)append(fp, n.file_ptr.size, n.file_ptr.data);
}
```

Listing 2: node_tree

The function *add* is responsively for inserting data. This is done with a key, provided to the b+ tree. See figure 5, case 1. This is an example where add is invoked with a key. Because the root has a node, add_recursive is invoked. The number of add_recursive invokes is based on the level, where e.g. level 2 would be equal to two recursive calls. In this case, the second recursive call is creating a new node due to its root has been splitted. This is returned to the first recursive call, and because its root has not been splitted, it returns null to the *add* function, which ultimaly return the new root.

In case two both recursively calls return a null due to no roots has been splitted. Ultimately the *add* function will return itself, see listing 3. Here add_custom is invoked with the key. Due to that add_custom_key is too large to display, please refer to appendix section 7.1.2 file inode.c, line 207 - 235, and 155 - 177 for add_recursive.

```
inode_t * add(inode_t * root, char * key, size_t value_size, void * value){
    return add_custom_key(root,strlen(key) + 1, key ,value_size,value);
}
```

Listing 3: add

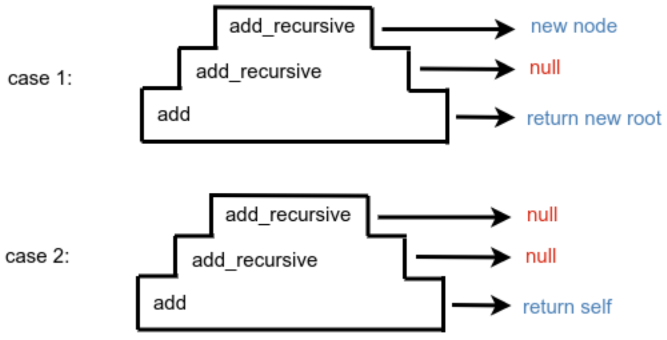

fig:5

The get function firstly checks if a node is a branch-node, or leaf-node. If it is a branch-node it checks if the key fits between variables in the node, and moves it to child-node that fits the interval. If it is a leaf-node, it checks if the key equal to a variabel in the node.

```
ssize_t get(inode_t * node, char * key){
  ssize_t index = find(node,key);
  if(0 > index) {
    return index;
  }
  ssize_t pos = node->values[index];
  if(node->level && 0 <= pos){
    node = struct_read(node->file_ptr.file , pos, *node);
    pos = get(node,key);
    free(node);
  }
  return pos;
}
```

Listing 4: get

The *tree_key* function returns the leftmost key of a the b+ tree. See figure 6. In this example, nodes can have at most two elements. The root has elements 1 and 2. Then 3 and 4 is inserted, hence, the node is splitted into two nodes. A new root is created, and to assign a key to the new root, *tree_key* is invoked. It searches untill it reaches the leftmost child, hence 3 is the new root key. Now 50, 100, 34 and 45 is inserted. Notice 34 is now the new key of yet another root.

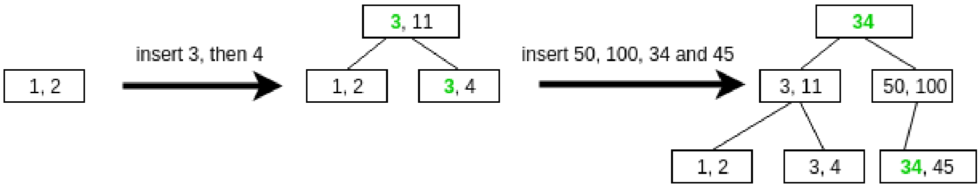

fig:6

```
file_ptr * tree_key(inode_t *node){
  file_ptr * f;
  if(node->level){
    inode_t * n = struct_read(
      node->file_ptr.file ,
      node->values[0] ,
      *n
    );
    f = tree_key(n);
    free(n);
  } else{
    f = get_string(
      node->file_ptr.file ,
      node->keys[0]
    );
  }
  return f;
}
```

Listing 5: tree_key

The delete function resembles the get-function a lot in the way it finds the element to be deleted. In figure 7, element b is to be deleted. This is done by saving the memory address of b, then remove its pointer, hence now the spot where b was is free. Finally move elements c's pointer to the free spot.

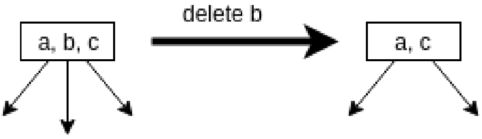

fig:7

```c
ssize_t delete(inode_t * node, char * key){
  ssize_t index = 0, pos = 0;
  index = find(node,key);
  pos = node->values[index];
  if(node->level && 0 <= pos){
    node = struct_read(node->file_ptr.file ,pos,*node);
    pos = delete(node,key);
    free(node);
  } else if(0 <= index){
    memmove(
      node->keys + index,
      node->keys + index + 1,
      (node->fill - index - 1) * sizeof(*node->keys)
    );
    node->keys[node->fill - 1] = 0;
    memmove(
      node->values + index,
      node->values + index + 1,
      (node->fill - index - 1) * sizeof(*node->values)
    );
    node->values[node->fill - 1] = 0;
    node->fill --;
    buf_write(&node->file_ptr);
  } else{
    pos = -1;
  }
  return pos;
}
```

Listing 6: delete

4 Test

Five test has been made to verify that the solution works correctly, and all requirements is meet.

The first test, as seen in the video at time-frame: 0:15 - 0:32, is to verify that its possible to create folders. First 'ls' is typed to show that the current working directory is empty. Then 'mkdir somefolder' is typed to create the folder, followed by another 'ls' command to verify it is created. To show that timestamps is also implemented, 'ls -alt' is typed too.

Test two, at time-frame 0:34 - 0:57, is to verify that its possible to create files, and add content to files. First 'cd somefolder' is typed to change directory into the newly created folder. Then 'echo "i like carrots" > rabbit.txt' is typed to create the file and its content. Then 'ls' to show the file. Then 'ls -alt' to verify the timestamp.

Test three, at time-frame 1:00 - 1:23, is to verify that its possible to read a file. With 'cat rabbit.txt' the file is read. To show its possible to append data 'echo ".. and beers" » rabbit.txt' is typed. Then 'cat rabbit.txt' is typed again, to read the updated file.

Test four, at time-frame 1:25 - 1:54, is to verify that its possible to delete files and folders. 'rm rabbit.txt' is typed to delete the file. Then 'ls' to verify its deleted. Then 'cd ..' to change to parent directory. Now 'rm -r somefolder/' is typed to delete the folder. Lastly 'ls' is typed to verify the deletion.

Test five, at time-frame 1:54 - 2:46, is to verify that its possible create multiple nested folders with files. A folder 'somefolder_again' is created, inside this folder, a command to recursively create four folders with three subfolders in each, and one file in all folders, is typed. Then 'tree' command is typed to visualize the folders and files. Lastly all the folders are deleted, and 'tree' is typed again, to verify everything is deleted.

5 Discussion

Our implementation has a very high garanty to recover from a powerfailure due to the fact the file is updated frequently with the data. The only time when data would be unrecoverable is if the powerfailure would happen during a write to file, and even then, only the actual string currently being written would be lost, everything else would be stored in the file.

6 Conclusion

A file-system in linux using FUSE has been developed. Its possible to create and list both folders and files, and delete them again. Furthermore its possible to open-, read-, write- and close files. Lastly its possible to show access- and modification time-stamps of files. Every requirements has been met.

7 Appendix

7.1 Source-code

7.1.1 header-files

file_ptr.h

```c
#ifndef DM510_FILE_PTR_H
#define DM510_FILE_PTR_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
  FILE * file;
  size_t pos;
  size_t size;
  char data[]; // <- Allows the expandsion of the struct.
} file_ptr;

// The end of the filestream, returns the size of the filestream.
size_t fend(FILE *);

// Reads a buffer of a size from a postion in a filestream.
file_ptr * buf_read(FILE *, size_t pos, size_t size);

// Writes a file pointer.
size_t buf_write(const file_ptr *);

// Appends a buffer at the end of the filestream.
file_ptr * append(FILE *, size_t size, const char * buf);

// Appends a file_ptr to the end of the filestream. OBS: updates the file pointers position.
int append_file_ptr(file_ptr * fp);

// Creates a new file_ptr, copies the buffer into itself.
file_ptr * new_file_ptr(FILE *fs, size_t pos, size_t size, const char * buf);

#define struct_read(file,pos,value) (typeof(value)*) buf_read(file,pos, sizeof(value) - sizeof(file_ptr))
#define primitiv_read(file,pos,value) ({file_ptr * fs = buf_read(file,pos, sizeof(value)); typeof(value) a ; memcpy(&a,fs->data,sizeof(a)); free(fs); a ;})
#define struct_write(value) buf_write((file_ptr *) value)
#define struct_append(file,value) (typeof(value)*) append(file, sizeof(value) - sizeof(file_ptr), (const char *)&value + sizeof(file_ptr))

#endif /* end of include guard: DM510_FILE_PTR_H */
```

fstruct.h

```c
#ifndef DM510_FSTRUCT_H
#define DM510_FSTRUCT_H
#include <time.h>
#include "inode_files/inode.h"

typedef struct{
  inode_t inode;
  mode_t type;
  char data[];
} group_t;


typedef struct{
  group_t group;
  size_t size;
  time_t access_time;  /* Time of last access */
  time_t modfication_time;  /* Time of last modification */
} file_t;

typedef struct{
  file_ptr file_ptr;
  char page[1 << 12];
} page_t;

#endif /* end of include guard: DM510_FSTRUCT_H */
```

inode.h

```c
#ifndef DM510_INODE_H
#define DM510_INODE_H
#include "file_ptr.h"
#define KEY_EXISTS -1
#define KEY_NOT_EXISTS -2
#define FULL_INODE -3
#define INODE_SIZE 4

typedef struct{
  file_ptr file_ptr;
  size_t level;
  size_t fill;
  size_t keys[INODE_SIZE];
  size_t values[1 + INODE_SIZE];
  char data[];
} inode_t;

//Initialise the N and B printf keywords.
void init_printf_inode_extension();

//Get a string from a given file and position
file_ptr * get_string(FILE *fs, size_t pos);

//Create a b+ tree of a given depth, OBS: only the left most part of the nodes are filled.
inode_t * node_tree(FILE *fs, size_t depth);

//Get the leftmost key of a given b+ tree.
file_ptr * tree_key(inode_t *);

//Add some data, with a key, to the given b+ tree. OBS: Should the root splt, the new root will returned from the function.
```

```
inode_t * add(inode_t *, char *, size_t, void *);

inode_t * add_custom_key(inode_t *,size_t, void *, size_t, void *);

//Get the data of a given key.
ssize_t get(inode_t * , char *);

//Delete a key and its accompanying data.
ssize_t delete(inode_t * , char *);

#define struct_add(node,key,st) add(node,key,sizeof(st),&st)
#define struct_add_fptr(node,key,st) add(node,key,sizeof(st) - sizeof(file_ptr), (char *)&st + sizeof(file_ptr))


#endif /* end of include guard: DM510_INODE_H */
```

lfs.h

```
#ifndef DM510_LFS_H
#define DM510_LFS_H

#include <fuse.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdio.h>

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_truncate( const char *, off_t, struct fuse_file_info *fi );
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_release(const char *path, struct fuse_file_info *fi);
int lfs_write( const char *, const char *, size_t, off_t, struct fuse_file_info *);
int lfs_utime( const char *, struct utimbuf *buf);

static struct fuse_operations lfs_oper = {
  .getattr  = lfs_getattr,
  .readdir  = lfs_readdir,
  .mknod = lfs_mknod,
  .mkdir = lfs_mkdir,
  .unlink = lfs_unlink,
  .rmdir = lfs_rmdir,
  .truncate = NULL,
  .open = lfs_open,
  .read = lfs_read,
  .release = lfs_release,
  .write = lfs_write,
  .rename = NULL,
  .utime = lfs_utime
};

```

```


#endif /* end of include guard: DM510_LFS_H */
```

7.1.2 source-files

file_ptr.c

```
#include "file_ptr.h"

file_ptr * new_file_ptr(FILE * fp, size_t pos, size_t size, const char * data){
  file_ptr * f = malloc(sizeof(*f) + size);
  f->file = fp;
  f->pos = pos;
  f->size = size;
  if(data) memcpy(f->data, (char*)data, size);
  return f;
}

size_t fend(FILE * fp){
  fseek(fp,0,SEEK_END);
  return ftell(fp);
}

file_ptr * buf_read(FILE * fp, size_t pos, size_t size){
  fseek(fp,pos,SEEK_SET);
  file_ptr * f = new_file_ptr(fp,pos,size,NULL);
  int written = fread(f->data, size, 1, fp);
  if(!written) {
    free(f);
    return NULL;
  }
  return f;
}

size_t buf_write(const file_ptr * fp){
  fseek(fp->file,fp->pos,SEEK_SET);
  return fwrite(fp->data,fp->size,1,fp->file);
}

file_ptr * append(FILE *fp, size_t size, const char * data){
  file_ptr *f = new_file_ptr(fp,fend(fp),size,data);
  int written = fwrite(f->data,f->size,1,f->file);
  if(!written) {
    free(f);
    return NULL;
  }
  return f;
}

int append_file_ptr(file_ptr * f){
  f->pos = fend(f->file);
  return fwrite(f->data,f->size,1,f->file);
}
```

inode.c

```c
#include "inode.h"
#include <string.h>
#include <printf.h>



void print_key(FILE *stream, const inode_t * bpt, size_t i){
  if(bpt->level){
    fprintf(stream, "%lu", bpt->keys[i] );
  } else {
    fprintf(stream, "(%lu)",bpt->keys[i] );
    if(bpt->keys[i]){
      file_ptr * string =  get_string(bpt->file_ptr.file,bpt->keys[i]) ;
      fprintf(stream, "%s", string->data );
      free(string);
    }
  }
}

int print_arginfo (const struct printf_info *info, size_t n, int *argtypes){
  /* We always take exactly one argument and this is a pointer to the structure.. */
  if (n > 0)
    argtypes[0] = PA_POINTER;
  return 1;
}

int print_inode(FILE *stream, const struct printf_info *info, const void *const *args){

  const inode_t * bpt = *((const inode_t **) (*args));
  int length = 0 ;
  size_t i;
  length += fprintf(stream, "%lu /%lu -> %lu : [",bpt->level,bpt->fill , bpt->file_ptr.pos );
  for (i = 0; i < INODE_SIZE - 1; i++) {
    print_key(stream, bpt, i);
    length += fprintf(stream, ", ");
  }
  print_key(stream, bpt,i);
  length += fprintf(stream, "], [%lu",bpt->values[0]);
  for (i = 1; i < INODE_SIZE + 1; i++) {
    length += fprintf(stream, ", %lu", bpt->values[i]);
  }
  length += fprintf(stream, "]");
  return length;
}

int print_bpr_recursive(FILE *stream, const inode_t * bpt, size_t index){
  const void *const arg = &bpt;
  int sum = print_inode(stream, NULL, &arg);
  if(!bpt->level) return sum;
  for (size_t i = 0; i < bpt->fill; i++) {
    inode_t * n = struct_read(bpt->file_ptr.file, bpt->values[i], *n);
    if(bpt->values[i]){
      sum += fprintf(stream, "\n%*s",(int)index + 1, " ");
      sum += print_bpr_recursive(stream, n, index + 1);
    }
    free(n);
  }
  return sum;
}

int print_bpr(FILE *stream, const struct printf_info *info, const void *
    const *args){
  const inode_t * bpt = *((const inode_t **) (*args));
  return print_bpr_recursive(stream,bpt,0);
}

void init_printf_inode_extension(){
  register_printf_function ('N', print_inode, print_arginfo);
  register_printf_function ('B', print_bpr, print_arginfo);
}

file_ptr * get_string(FILE *fp, size_t pos){
  size_t length = 1;
  fseek(fp,pos,SEEK_SET);
  while(fgetc(fp)) length++;
  return buf_read(fp,pos,length);
}

file_ptr * tree_key(inode_t *node){
  file_ptr * f;
  if(node->level){
    inode_t * n = struct_read(
      node->file_ptr.file,
      node->values[0],
      *n
    );
    f = tree_key(n);
    free(n);
  } else{
    f = get_string(
      node->file_ptr.file,
      node->keys[0]
    );
  }
  return f;
}

ssize_t match(inode_t * node, char * key){
  const size_t size = node->fill - (node->level > 0);
  size_t i = 0;
  for (; i < size; i++) {
    file_ptr * string = get_string(node->file_ptr.file, node->keys[i]);
    int cmp = strcmp(key,string->data);
    free(string);
    if(0 > cmp){
      return i;
    } else if(0 == cmp){
      return node->level ? i + 1 :  KEY_EXISTS;
    }
  }
  return i;
}

inode_t * split_inode(inode_t * src, long index){
  inode_t * dest = calloc(1,sizeof(*dest));
  const size_t fsize = src->fill / 2, csize=(src->fill + 1) / 2;
  size_t size = src->level ? 1 : 0;
  dest->fill = fsize;
  dest->file_ptr = src->file_ptr;
  dest->level = src->level;
  src->fill = csize;
  size_t type = sizeof(*src->keys);
  memcpy(dest->keys, src->keys + csize, (fsize - size) * type);
  memset(src->keys + fsize, 0 ,fsize * type);

  type = sizeof(*src->values);
  memcpy(dest->values, src->values + csize, (fsize ) * type);
  memset(src->values + csize, 0 ,fsize * type);
  return dest;
}

void node_make_room(inode_t * node,long index){
  size_t size = (node->fill - index - (node->level > 0)) * sizeof(*node->keys);
  memmove( node->keys + index + 1, node->keys + index, size);

  size = (node->fill - index) * sizeof(*node->values);
  memmove(node->values + index + 1, node->values + index, size);
}

inode_t * node_tree(FILE *fp, size_t level){
  inode_t n = {
    .file_ptr.size = sizeof(n) - sizeof(file_ptr),
    .level = level,
    .fill = level > 0
  };
  if(level){
     inode_t * result = node_tree(fp,level - 1);
     n.values[0] = result->file_ptr.pos;
     free(result);
  }
  return (inode_t *)append(fp, n.file_ptr.size, n.file_ptr.data);
}


inode_t * add_recursive(inode_t * node, file_ptr * key, file_ptr * value){
  inode_t * out_node = node, *new_node = NULL, *right_node = NULL;
  ssize_t index = match(node,key->data);
  int new_item = 0, split = 0;

  if(0 > index) return new_node;
  if(node->level){
    if(!node->values[index]){ // Make new inode tree.
      right_node = node_tree(node->file_ptr.file, node->level);
    }else {
      right_node = struct_read(
        node->file_ptr.file,
        node->values[index],
        *right_node
      );
    }
    value = (file_ptr*)add_recursive(right_node,key,value);
    free(right_node);
    if(value){ //Handle split.
      key = tree_key((inode_t*)value);
      split = INODE_SIZE < node->fill;
      new_item = 1;
    }
  } else {
    split = INODE_SIZE == node->fill;
    new_item = 1;
  }
  if(split){
    new_node = split_inode(node, index);
    append_file_ptr(&new_node->file_ptr); // Write the old node.
    buf_write(&node->file_ptr); // Write the new node.
    if(index > node->fill){
      out_node = new_node;
    }
  }
  if(!node->level || new_item){
    index = match(out_node,key->data);
    if(index < out_node->fill){
      node_make_room(out_node,index);
    }
    out_node->values[index + (node->level > 0)] = value->pos;
    out_node->fill++;
    out_node->keys[index] = key->pos;
  }
  struct_write(out_node);
  return new_node;
}

inode_t * add(inode_t * root, char * key, size_t value_size, void * value){
    return add_custom_key(root,strlen(key) + 1, key ,value_size,value);
}

inode_t * add_custom_key(inode_t * root, size_t key_size,  void * key, size_t value_size, void * value){
  if(!(root && key && value)) return NULL;
  size_t size = sizeof(inode_t) - sizeof(file_ptr);

  file_ptr * key_ptr = append(root->file_ptr.file, key_size, key);
  file_ptr * value_ptr = append(root->file_ptr.file, value_size, value);

  inode_t * right_root = add_recursive(root,key_ptr,value_ptr);
  if(right_root){  // Root got split.
    inode_t * new_root = (inode_t *)new_file_ptr(root->file_ptr.file,0,size,NULL);
    memset(new_root->file_ptr.data,0,size);
    key_ptr = tree_key(right_root);
    new_root->keys[0] = key_ptr->pos;
    new_root->file_ptr = root->file_ptr;
    append_file_ptr(&root->file_ptr);

    new_root->level = root->level + 1;
    new_root->values[0] = root->file_ptr.pos;
    new_root->values[1] = right_root->file_ptr.pos;
    new_root->fill = 2;

    buf_write((file_ptr*)new_root);
    free(right_root);
    root = new_root;
  }
  free(key_ptr);
  free(value_ptr);
  return root;
}

```

```c
ssize_t find(inode_t * node, char * key){
  size_t i = 0;
  size_t size = node -> fill - (node->level > 0);
  for (; i < size; i++) {
    file_ptr * string = get_string(node->file_ptr.file , node->keys[i]);
    int cmp = strcmp(key,string->data);
    free(string);
    if((node->level && 0 > cmp) || (!node->level && !cmp)){
      return i;
    }
  }
  return node->level ? i : KEY_NOT_EXISTS;
}

ssize_t get(inode_t * node, char * key){
  ssize_t index = find(node,key);
  if(0 > index) {
    return index;
  }
  ssize_t pos = node->values[index];
  if(node->level && 0 <= pos){
    node = struct_read(node->file_ptr.file , pos, *node);
    pos = get(node,key);
    free(node);
  }
  return pos;
}

ssize_t delete(inode_t * node, char * key){
  ssize_t index = 0, pos = 0;
  index = find(node,key);
  pos = node->values[index];
  if(node->level && 0 <= pos){
    node = struct_read(node->file_ptr.file ,pos,*node);
    pos = delete(node,key);
    free(node);
  } else if(0 <= index){
    memmove(
      node->keys + index,
      node->keys + index + 1,
      (node->fill - index - 1) * sizeof(*node->keys)
    );
    node->keys[node->fill - 1] = 0;
    memmove(
      node->values + index,
      node->values + index + 1,
      (node->fill - index - 1) * sizeof(*node->values)
    );
    node->values[node->fill - 1] = 0;
    node->fill --;
    buf_write(&node->file_ptr);
  } else{
    pos = -1;
  }
  return pos;
}
```

lfs.c

```c
#include "lfs.h"
#include "fstruct.h"

#define min(x,y) x < y ? x : y
#define max(x,y) x > y ? x : y


FILE * file_system;

typedef struct{
  char* string;
  char* end;
} split_path_t;

split_path_t split_path(const char * path, char * (*f)(const char *, int)){
  split_path_t p;
  const char * end = f(path + 1, '/');
  const char * pad = path + ('/' == *path);
  if(!end){
    size_t size = strlen(pad);
    p.end = calloc(1, size + 1);
    strncpy(p.end, pad, size);
    p.string = NULL;
  }else{
    p.string = calloc(1, end - pad + 1);
    strncpy(p.string , pad, end - pad);
    const size_t size = strlen(end);
    p.end = calloc(1, size + 1);
    strncpy(p.end, end + 1,size);
  }
  return p;
}

group_t * walk(group_t * group, const char *path){
  if(!path) return NULL;

  split_path_t sp = split_path(path, strchr);
  ssize_t pos = get(&group->inode, sp.string ? sp.string : sp.end);
  if(0 > pos){
    free(sp.string);
    free(sp.end);
    return NULL;
  }
  group_t * g = struct_read(group->inode.file_ptr.file , pos, *g);
  if(sp.string){
    group = walk(g,sp.end);
    free(g);
  } else{
    group = g;
  }
  free(sp.string);
  free(sp.end);
  return group;
}


int lfs_getattr( const char *path, struct stat *stbuf ) {
  int res = 0;
  group_t * root = struct_read(file_system, 0, *root);

  memset(stbuf, 0, sizeof(struct stat));
  if( strcmp( path, "/" ) == 0 ) {
    stbuf->st_mode = S_IFDIR | 0755;
    stbuf->st_nlink = 2;
  } else  {
    group_t * group = walk(root,path);
    if( group ){
      stbuf->st_mode = group->type | 0777;
      stbuf->st_nlink = 2;
      if( S_IFREG == group->type) {
        stbuf->st_nlink = 1;
        file_ptr * fp = &group->inode.file_ptr;
        file_t * file =  struct_read(fp->file, fp->pos, *file);
        stbuf->st_size = file->size;
        stbuf->st_atime = file->access_time;
        stbuf->st_mtime = file->modfication_time;
      }
      free(group);
    }else{
      res = -ENOENT;
    }
  }
  free(root);
  return res;
}


void lfs_list_dir(inode_t * node, void *buf, fuse_fill_dir_t filler){
  if(node->level){
    for (size_t i = 0; i < node->fill; i++) {
      inode_t * n = (inode_t*)buf_read(
        node->file_ptr.file,
        node->values[i],
        node->file_ptr.size
      );
      lfs_list_dir(n,buf,filler);
      free(n);
    }
  }else{
    for (size_t i = 0; i < node->fill; i++) {
      file_ptr * string = get_string(node->file_ptr.file, node->keys[i]);
      printf("string(%lu) = %s\n",node->keys[i], string->data );
      filler(buf,string->data,NULL,0);
      free(string);
    }
  }
}

int lfs_readdir( const char *path, void *buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi ) {

  printf("readdir: (path=%s)\n", path);
  filler(buf, ".", NULL, 0);
  filler(buf, "..", NULL, 0);

  group_t * root = (group_t *)buf_read(file_system, 0, sizeof(group_t) - sizeof(file_ptr));
  group_t * node = walk(root,path);
  if(node){
    lfs_list_dir(&node->inode,buf,filler);
    free(node);
  } else{
    lfs_list_dir(&root->inode,buf,filler);
  }
  free(root);
  return 0;
}

int lfs_mknod( const char * path, mode_t mode, dev_t dev){
  group_t * root = struct_read(file_system, 0, *root );
  split_path_t sp = split_path(path,strrchr);
  group_t * spot = walk(root,sp.string);
  if(spot){
    free(root);
    root = spot;
  }
  file_t file = {
    .group.type = S_IFREG,
    .size = 0,
    .access_time = time(NULL),
    .modfication_time = time(NULL)
  };
  struct_add_fptr(&root->inode, sp.end, file);
  free(sp.string);
  free(sp.end);
  return 0;
}

int lfs_mkdir( const char * path, mode_t mode){
  group_t * root = struct_read(file_system, 0, *root);
  split_path_t sp = split_path(path, strrchr);
  group_t * spot = walk(root, sp.string);
  if(spot){
    free(root);
    root = spot;
  }
  group_t group = {.type = S_IFDIR};

  struct_add_fptr(&root->inode, sp.end, group);
  free(root);
  free(sp.string);
  free(sp.end);
  return 0;
}
int lfs_unlink(const char * path){
  group_t * root = (group_t *)buf_read(file_system, 0, sizeof(group_t) -
    sizeof(file_ptr));
  split_path_t sp = split_path(path,strrchr);
  group_t * spot = walk(root,sp.string);
  if(spot){
    free(root);
    root = spot;
  }
  delete(&root->inode,sp.end);
  free(sp.string);
  free(sp.end);
  return 0;
}

int lfs_rmdir( const char * path){
  group_t * root = (group_t *)buf_read(file_system, 0, sizeof(group_t) -
```

```c
    sizeof(file_ptr));
  split_path_t sp = split_path(path,strrchr);
  group_t * spot = walk(root,sp.string);
  if(spot){
    free(root);
    root = spot;
  }
  delete(&root->inode,sp.end);
  free(sp.string);
  free(sp.end);
  return 0;
}

int lfs_open( const char *path, struct fuse_file_info *fi ) {
  return 0;
}

char * hash(size_t key){
  char *string = calloc(1,9);
  sprintf(string,"%08x", (unsigned int)key);
  return string;
}

page_t * merge_page(inode_t ** node, size_t key){
  char * key_hash = hash(key);
  ssize_t pos = get(*node, key_hash);
  if(0 > pos){
    static page_t p;
    *node = struct_add(*node, key_hash, p);
    pos = get(*node, key_hash);
  }
  free(key_hash);
  return struct_read((*node)->file_ptr.file , pos, page_t);
}

int lfs_read( const char *path, char *buf, size_t size, off_t offset, struct
     fuse_file_info *fi ) {
  group_t * root = struct_read(file_system, 0, *root);
  group_t * spot = walk(root,path);
  free(root);
  file_ptr *fp = &spot->inode.file_ptr;
  file_t * file = struct_read( fp->file , fp->pos,* file );
  free(spot);

  const size_t start = offset , end = start + size;
  while(offset < end) {
    const off_t local_offset = offset % sizeof(page_t);
    const size_t local_size = min(end - offset , sizeof(page_t) -
    local_offset);

    inode_t * node = &file->group.inode;
    page_t * page = merge_page(&node, offset / sizeof(page_t) );
    memcpy(offset - start + buf, page->page + local_offset , local_size);
    offset += local_size;
    free(page);
  }
  file->access_time = time(NULL);
  struct_write(file);
  return size;
}

```

```c
int lfs_write( const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi){
  group_t * root = struct_read(file_system, 0, *root);
  group_t * spot = walk(root,path);
  free(root);
  file_ptr *fp = &spot->inode.file_ptr;
  file_t * file = struct_read( fp->file, fp->pos,*file );
  free(spot);

  const size_t start = offset, end = start + size;
  while(offset < end) {
    const off_t local_offset = offset % sizeof(page_t);
    const size_t local_size = min(end - offset, sizeof(page_t) - local_offset);

    inode_t * node = &file->group.inode;
    page_t * page = merge_page(&node, offset / sizeof(page_t) );
    memcpy(page->page + local_offset, offset - start + buf ,local_size);

    offset += local_size;
    file->size += local_size;

    struct_write(page);
    free(page);
  }
  file->modfication_time = time(NULL);
  struct_write(file);
  return size;
}

int lfs_release(const char *path, struct fuse_file_info *fi) {
  return 0;
}

int lfs_utime( const char * path, struct utimbuf *buf){
  return 0;
}

FILE * merge(char * file, char * mode){
  FILE * fp = fopen(file,mode);
  if(!fp) {
    fp = fopen(file,"w");
    fclose(fp);
    fp = fopen(file,mode);
  }
  return fp;
}

int main( int argc, char *argv[] ) {
  init_printf_inode_extension();
  file_system = merge("filesystem.fs","r+b");
  size_t size = sizeof(group_t) - sizeof(file_ptr);
  group_t * root = (group_t *)buf_read( file_system, 0, size);
  if(!root){
    printf("No root!\n");
    group_t empty = {.type = S_IFDIR};

    root = struct_append(file_system, empty);
  }
  printf("%B\n", root );
  fuse_main( argc, argv, &lfs_oper );
  return 0;
}
```