INTRODUCTION TO OPERATING SYSTEM

PROJECT ON

FILE SYSTEM

SHADAN ALAM KAIFEE
(01FB16ECS346)
SHASHANK SHEKHAR PATHAK
(01FB16ECS359)
SHASHWAT MISHRA
(01FB16ECS361)
SHIKHAR SHRESTHA
(01FB16ECS362)

(CSE-5TH SEMESTER - ‘F’)

Overview

This is a simple file-system implemented using FUSE(File System in User Space), using a single large file(backing storage) as an emulator for a block-oriented disk. In this report, we explain the design of our project, including any relevant considerations and decisions. We explain the architecture of the project, its organization, and how its parts come together. Finally, we conclude with our observations, as well as some of the limitations of our work.

Goals

The purpose of this project was to gain practical knowledge of Linux internals by creating a basic filesystem using tree structure to simulate hierarchical representation.

What is FUSE?

Fuse is a software interface for Linux like computer systems that allows non-privileged users to create their own filesystem without modifying kernel code. The resulting file system resides in the userspace and exists as a layer of abstraction over a more concrete and robust filesystem.

How is our File System Implemented?

Phases Implementation:

Phase I: System Call Implementation

FUSE was used as an interface to implement the system calls, since it consists of prototypes for various system calls(specified in “fuse.h”), and the same were registered using its “operations” structure.

Phase II: File System Abstractions

Implement file abstractions: block management, inode structure, data blocks, directory structure (if not done in phase 1). Port your file system into secondary memory. We have implemented openblock, readblock, writeblock as given in the manual and integrated system call implementation from phase 1 with file abstraction (phase 2).

Phase III: Secondary Storage

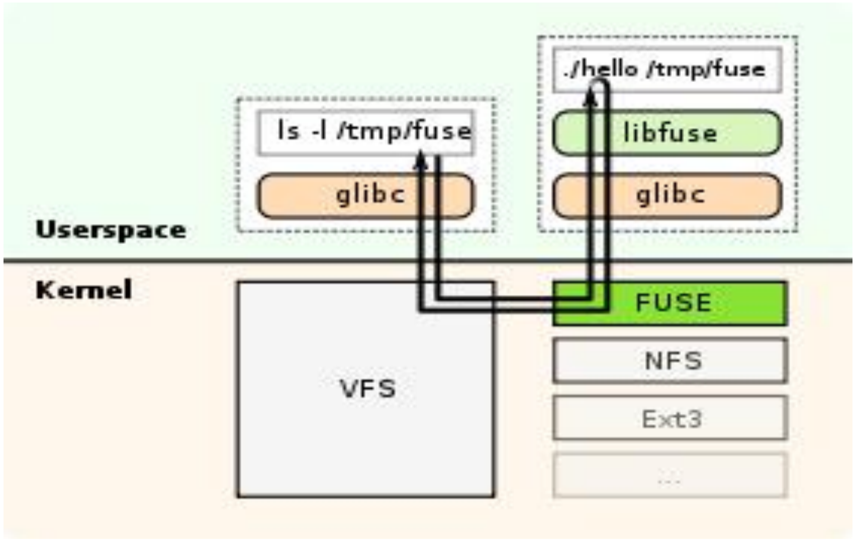
We have implemented the filesystem using a single large file as the secondary storage so as to achieve persistance. We store the actual data and the meta data for the entire file system into the secondary storage when we unmount the filesystem. On remounting, we fetch the data back into the tree structure of the file system from the secondary storage.

Phase I

We start by explaining in depth working of FUSE.

To implement a new file system, a handler program linked to the supplied libfuse library needs to be written. The main purpose of this program is to specify how the file system is to respond to read/write/stat requests. The program is also used to mount the new file system. At the time the file system is mounted, the handler is registered with the kernel. If a user now issues read/write/stat requests for this newly mounted file system, the kernel forwards these IO-requests to the handler and then sends the handler's response back to the user. These handlers are registered using fuse’s structure “fuse_operations”.

FUSE is particularly useful for writing virtual file systems. Unlike traditional file systems that essentially work with data on mass storage, virtual filesystems don't actually store data themselves. They act as a view or translation of an existing file system or storage device.



Functions Implemented

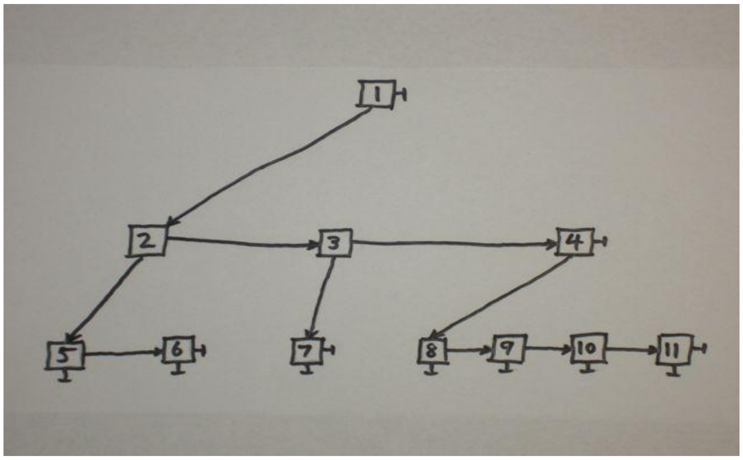
1. **int fsys_mkdir(const char *path, mode_t mode):**Creates a new directory by adding the “path” to the directory table with default mode set to 0755.
2. **int fsys_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset):**Reads the entries in a directory and fills the buffer provided by the caller.
3. **int sys_open(const char *path):**Opens a file in the given path.
4. **int sys_rmdir(const char *path):**It removes the given empty directory and remove the path from Dir_table.
5. **int sys_unlink(const char *path):**It is used by rm , it decrease the nlink count of a file in the stat struct.
6. **int fsys_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info*fi):**reads from a file and stores it onto “buf” and returns the number of bytes/size read.
7. **int fsys_getattr(const char *path, struct stat *stbuf):**initializes the default attributes of a file/directory especially the root(when the FS mounts).
8. **int fsys_create(const char *path, mode_t mode, struct fuse_file_info *fi):**Creates a new file with default mode set to 0644.
9. **static int fsys_write(const char *path, const char *buf, size_t size, off_t offset, struct**
   fuse_file_info *fi): writes “size” amount of data from the “offset” specified from “buf” onto the file.
10. **static int fsys_rename(const char * from, const char * to):** renames a file to the “to” parameter specified and internally calls fsys_rmdir(for a directory) and fsys_unlink (for a file).

Here is an image depicting the handlers registered in fuse_operations:

```
static struct fuse_operations hello_oper = {
    .getattr    = fsys_getattr,
    .readdir    = fsys_readdir,
    .open       = fsys_open,
    .read       = fsys_read,
    .utimens        = fsys_utime,
    .rmdir      = fsys_rmdir,
    .mkdir      = fsys_mkdir,
    .create         = fsys_create,
    .write          = fsys_write,
    .unlink     = fsys_unlink,
    .rename         = fsys_rename,
    .destroy    = fsys_destroy,
    .opendir    = fsys_opendir,
};
```

Phase II:

The File system implemented had the following similar Tree Hierarchy:-



The Above given diagram shows our file system structure . The file system is implemented as an n-ary tree in which a parent has a single child pointer to a linked list of childs and the children in turn point to their corresponding parent. E.g - In the above diagram the Linked list 2 -> 3 -> 4 are childs of Parent Node 1. Our actual Tree Inode is implemented as given below:

```
typedef struct __data {
      char name[MAX_NAME];
      int  isdir;
      struct stat st;
} Ndata;

typedef struct element {
      Ndata data;
      char * filedata;
      struct element * parent;
      struct element * firstchild;
      struct element * next;
} Node;
```

The Given Tree inode structure contains following:

1. Struct node element
   a. This structure is common for both files and directories.
   b. **Ndata data** :- This field is a metadata for the current file/directory
   c. **Char * filedata**:- This is pointer to the actual file/directory contents.
   d. **Struct element *parent** :- This is a pointer to the parent of the current child file/directory

e. **Struct element *firstchild**:- This pointer is present in Parent node which points to the linked list of it's children.

f. **Struct element *next** :- This is a pointer which points to the next child in the linked list

2. Struct __data

   a. This structure contains the metadata about the file/directory

   b. **Char name**:- This field has the name of the file/directory

   c. **Int isdir**:- This field indicates whether the node is file or a directory

   d. **Struct stat st**:- This field contains other necessary information for the node created like block size, uid ,gid ,access time etc.

Some key aspects of the each node in the file system is mentioned below:-

1. Each Node (directory) in the file system has a default size of 4096 bytes and each Node (file) has a default size of 0 byte at the time of creation. Memory is allocated in blocks of 4096 bytes boundary and if this threshold is crossed by any file/directory then another block of 4096 bytes are further allocated. A Default maximum size of the file system is 15 MB. Whenever a new Node is created a check for available free memory is done after which allocation proceeds if sufficient memory is available otherwise an error message is displayed.
2. The metadata is maintained in a tree structure, where every directory node is capable of storing any number of children, whereas file nodes are leaf nodes of the tree, and do not have any children.

Phase III:

The FS implemented in Phase I was non - persistent i.e. whenever we unmounted our FS, the data would get erased thus we used a file to store all the data of our FS before unmounting.

How unmounting actually occurs:

The destroy function:

void fsys_destroy(void* private_data);

Whenever we execute a command for unmounting our FS, FUSE has an inbuilt function called "destroy" that gets called, which performs clean - ups (including, erasing contents of our FS), thus the reason for our FS being non - persistent.

```
void fsys_destroy(void* private_data) {
        if  (filedump[0] == '\0') {
                return;
        }
        diskfile = fopen(filedump, "w+b");
        if (diskfile) {
                //write DS to disk
                fwrite(&Root->data, sizeof(Ndata), 1, diskfile);
                serialize(Root);
                fclose(diskfile);
        }
}
```

How we implemented persistence:

A brief summary:

> Whenever our FS got mounted the main function will check whether the file specified exists or not, if it does not exist that means the FS is being mount for the first time and thus normally allocated memory as well as various important information related to ROOT. If the file existed a deserialization algorithm was invoked that wrote the contents of FS from file back to the tree allowing persistence.
>
> As mentioned about destroy, during unmounting it will call a serialization algorithm that will write contents of every node of our tree onto the file (for persistence) in the form of bytes in a preorder fashion.

Thus a major role was played by the tree serialization - deserialization algorithm which has the following workflow:

```
int main(int argc, char *argv[])
{

    freememory = 50 * 1024 * 1024;  //provide max memory size available
    if (freememory <= 0) {
        fprintf(stderr, "Invalid Memory Size\n");
        return -1;
    }
    int init_done = 0;

        strncpy(filedump, "/home/hduser/Desktop/empty/storage2", MAX_NAME);
        diskfile = fopen(filedump,"rb");
        if (diskfile) {
            // Read from disk into ds
            allocate_node(&Root);
            fread(&Root->data, sizeof(Ndata), 1, diskfile);
            deserialize(Root);
            init_done = 1;
            fclose(diskfile);
```

A serialisation algorithm writes the contents of a tree onto a file(usually in the form of bytes).

The Tree Serialization Algorithm:

```
void serialize(Node * parent) {

        //fprintf(stderr, "%s\n",parent->data.name);
        int num_child = parent->data.st.st_nlink - 2;
        int i = 0;
        Node * temp = parent->firstchild;
        for (;i<num_child; i++) {
                fwrite(&temp->data, sizeof(Ndata), 1, diskfile);
                if (temp->data.isdir) {
                        serialize(temp);
                } else {
                        int filelen = temp->data.st.st_size;
                        fwrite(temp->filedata, sizeof(char), filelen, diskfile);
                }
                temp = temp->next;
        }
}
```

Now when, we remounted our FS specifying the same file (to which we previously wrote our data to), the initialization happens by checking if that file exists, since it does exist, our code invokes a function that implements a tree deserialization algorithm whose job is to convert data stored onto a file to a tree representation thus making it persistent across reboots and unmounts.

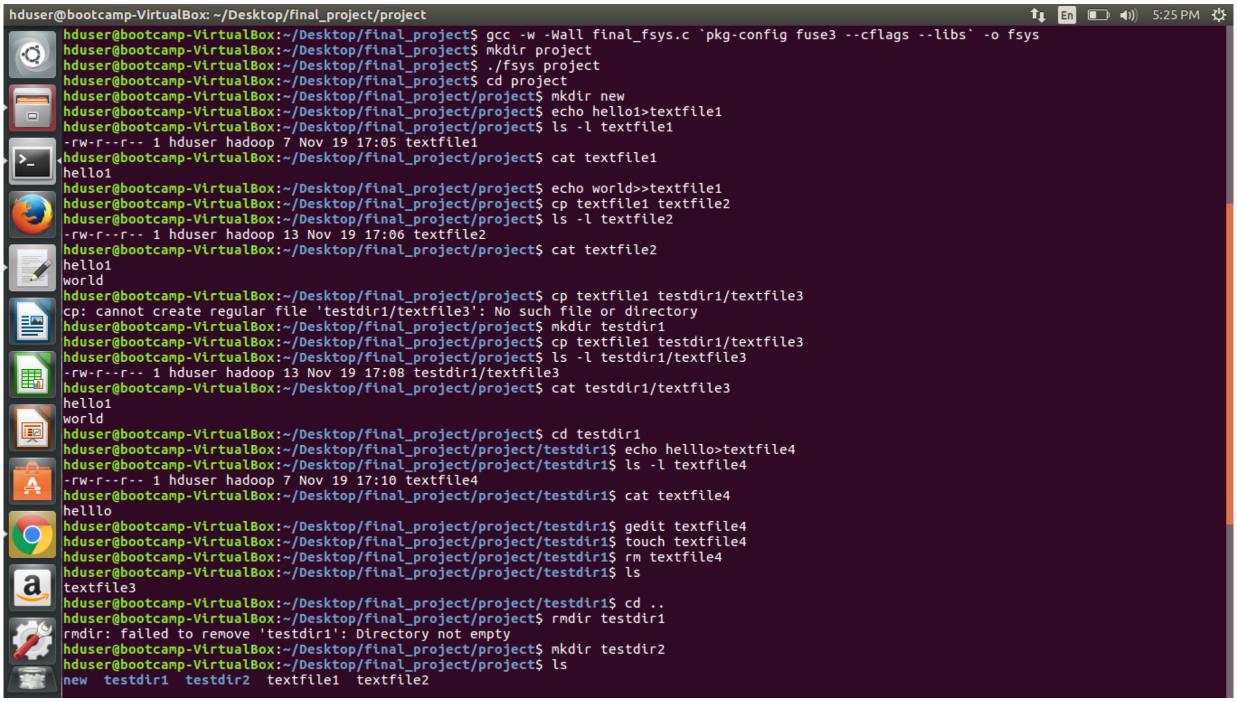
The tree deserialization algorithm:

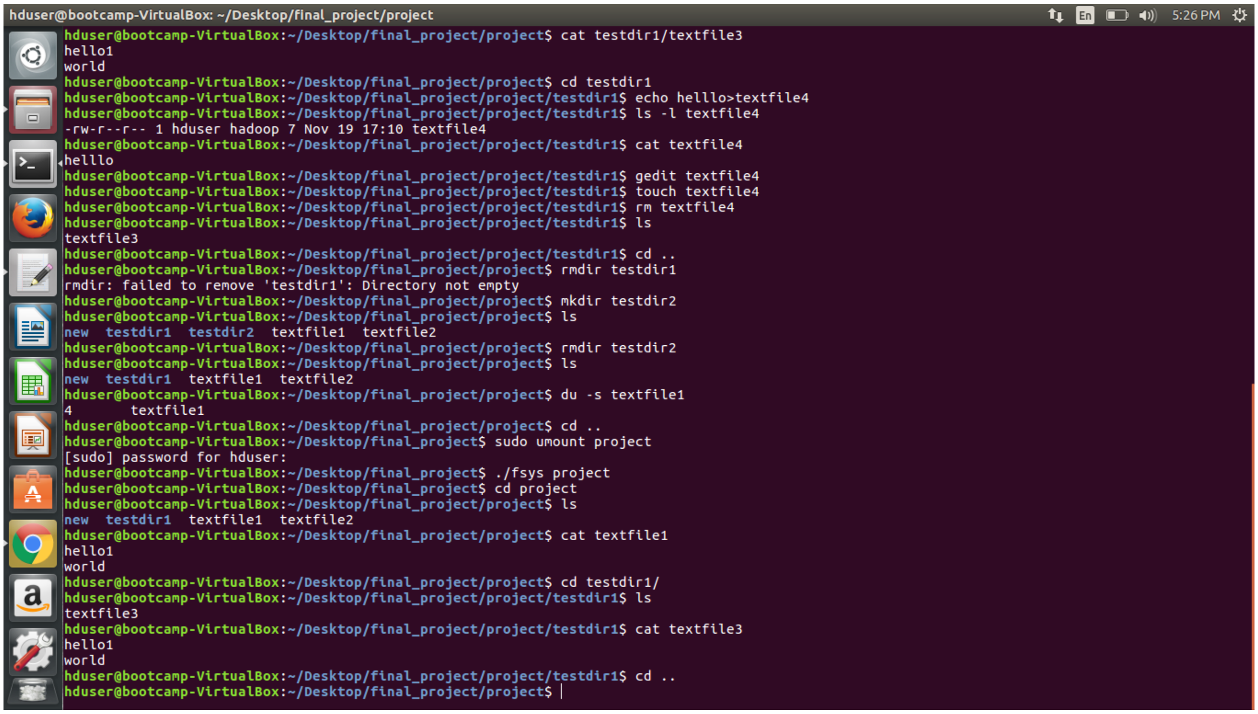
13

```
void deserialize(Node * parent) {
        int num_child = parent->data.st.st_nlink -2;
        int i;
        Node * x;
        Node * cur;
        if (num_child == 0) {
                return;
        }
        allocate_node(&x); parent->firstchild = x; x->parent = parent; cur = x;
        for (i=1; i< num_child; i++) {
                Node * y;
                int ret = allocate_node(&y);
                if (ret != 0) {
                        fprintf(stderr,"deserialize: No space left on device\n");
                        return;
                }
                cur->next = y;  y->parent = parent; cur = y;
        }
        Node * temp = parent->firstchild;
        for (i=0; i<num_child; i++) {
                fread(&temp->data, sizeof(Ndata), 1, diskfile);
                if (temp->data.isdir) {
                        deserialize(temp);
                } else {
                        int filelen = temp->data.st.st_size;
                        if (filelen > freememory) {
                                fprintf(stderr,"deserialize: No space left on device\n");
                                return;
                        }
                        temp->filedata = calloc(filelen, sizeof(char));
                        if (temp->filedata == NULL) {
                                fprintf(stderr,"deserialize: Not enough memory\n"); return;
                        }
                        freememory -= filelen;  fread(temp->filedata, sizeof(char), filelen, diskfile);
                }
                temp = temp->next;
        }
}
```

Test Run

Our filesystem worked successfully for the following commands:





Conclusion

In this project, we have created a basic filesystem, and a single large file as an emulator for a disk. It was a fun learning experience, as we were able to have more clear understanding of the internals of the file system.

However, our file system has still some flaws, as listed below, which can be improved upon:

- The search for the filepath is slow,because as the tree increases traversing the tree(i.e pointers to the node) becomes slow towards the leaf nodes.
- The system is not very performant, largely due to string matching times.
- Replacing the expensive prefix matching operations with file handles would probably solve most of the issue, however there will be limits due to the fact that the system is implemented on a file managed by another filesystem.

Bibliography

1. https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/f use_doc.html
2. http://www.maastaar.net/fuse/linux/filesystem/c/2016/05/21/writing-a-simple-f ilesystem-using-fuse/
3. https://www.unixtutorial.org/2008/04/atime-ctime-mtime-in-unix-filesystems/
4. http://www.geeksforgeeks.org/serialize-deserialize-n-ary-tree/
5. https://linux.die.net/man/8/mkfs
6.