Project 3 Report
Name: Xiaoyu_Ma Student id: 523031910469

1 Introduction

In this Report, I will describe the design and implementation of the File system, as well as the evaluations and tests Including the breakdown of the project, which are:

- Design a basic disk-storage system.
- Build a simple File system on top of the disk-storage system.
- Develop the system to support multiple users and clients, as well as file permissions management.
- Some bonus features.

The 3 parts are implemented in the order of complexity, and the first part is the most basic part of the project. The second part is to build a file system on top of the disk-storage system, which is a more complex part. The third part is to develop the system to support multiple users and clients, as well as file permissions management, which is the most complex part of the project. The bonus features are optional and can be implemented if time permits.

2 Disk Storage System

In this part, I was required to implement the simulated disk system, as an Internet-domain socket server, which is a simulation of a physical disk. The simulated disk is organized by cylinder and sector, just like a real disk.

2.1 Design

2.1.1 Manipulate the actural storage

The code Implemented Manipulate the actural storage with mmap() function, which is a system call that maps files or devices into memory. After using mmap(), the file is mapped into memory, and the pointer to the mapped memory is returned. The pointer can be used to read and write data to the file as if it were a normal array. Every time the file is modified, the changes are automatically written to the file on disk. This function gives a lot convience in the implementation of the project.

2.1.2 Handle the commands

The commands of the disk system are the following 3:

- I: information request, show the cylinder and sector of the disk.
- R c s: Read request for the contents of cylinder c sector s. The disk returns Yes followed by a whitespace and those 512 bytes of information, or No if no such block exists. The command will returns what ever there is in the block, even if it is not a valid file system block.
- W c s l data: Write a request for cylinder c sector s. l is the number of bytes being provided, with a maximum of 512. The data is those l bytes of data.

  - The disk returns Yes to the client if it is a valid write request or returns a No otherwise.
  - In cases where l < 512, the contents of those bytes of the sector between byte l and byte 512 are undefined, use zero-padded contents.

As the location to read or write is specified by the cylinder and sector, and we use mmap() to control the file on disk, the implementation of the disk system becomes clear: we just use the cylinder and sector number to calculate the offset of the file, and then use the pointer returned by mmap() to read or write the data.
The mmap() function has provided a lot of convenience, we just neet to use memcpy() to write or read contents.
Furthermore, to implement the fault Handlings, we need to store the metadata of the disk system, which includes the cylinders number and the sectors number per cylinder. These metadata are used to determine wheather the read or write request is valid, and they will be stored in the superblock of the file system, which we will discuss later.
Follow the instructions and the analysis, the development becomes clear.

2.1.3 implement the disk server and client

The disk server is implemented as a TCP socket server, using the provided socket library utilities, The server and client was implemented.

- ./BDS <DiskFileName> <#cylinders> <#sector per cylinder> <track-to-track delay> <port=10356>
- ./BDC <DiskServerAddress> <port=10356>

These commands will run disk server and clients respectively on localhost.

3 Design a basic file system

In the section above, I have implemented the storage system, which is a simulated disk system. In this section, I will implement a file system on top of the disk system. Here are the breakdown of the development.

3.1 Inode system

Like linux fule system, this file system project is also designed to use iNode to organized the storage. The iNode stores either the information data of the file, and the disk block address where the data settles in. The iNode contains the following information:

- The file type: Directory or file.
- The file size: The size of the file in bytes.
- The block number: how many blocks are used Directly to store file. Note that in the design it will never decrease.
- The name and permissions, owner of the file.
- Direct links and indirect links to the data blocks.
- inum: which is the index of iNode in disk system.

- The time of last access and modification.

To simplify the implementation, all users are identified solely by their UID (without usernames) and can access the system without passwords. The file access protocol is simplified to 2 groups:

- Owner: The owner of the file, Whose permissions are controlled by the first 3 bit of the permissions code.
- other: All other users, who have read-only access to the file., or determined by the file permissions.
- Root: user 1. The root user has all permissions on the file, and can access any file in the system.

Then we can implement the iNode structure, to flexibly store the information of the file, the iNodes are divided into 2 types:

- disk iNode: The iNode is stored in the disk system.
- memory iNode: The iNode is stored in the memory. Each time, the memory iNode will load data from corresponding disk iNode, and data will be synchronized to the disk iNode when the memory iNode is freed.

Then we design some basic operations on the iNode, such as iput, ialloc, iget, iput, iupdate, iwrite, iread, and so on. Whats more, some disk operations functions like allocation of iNode blocks are alose needed.

3.2 File system design

In this part, I wiil Introduce the functions for basic file system commands, such as ls, cd, mkdir, rmdir and so on.

- f: Format. This will format the file system on the disk, by initializing any/all of the tables that the file system relies on.
- ls: list the files in the current directory.
- cd: change the current directory.
- mkdir: create a new directory.
- rmdir: remove a directory.
- rm: remove a file.
- w f l data: Write data into file
- i f pos l data: Insert data to a file at position pos
- d f pos l: Delete data in the file. This will delete l bytes starting from the pos character (0-indexed), or till the end of the file
- e: exit the file system.

All operations on the Directories and files are implemented based on iNode, that is, we regard the directories as a special type of file, the contents of those directories are the pointers to the file iNodes.

3.2.1 f command

This command is used to format the file system, which will initialize the superblock, and the iNode table, as well as wipe out the disk system. It is handled by the cmd_f() function. Each time this is called, the superblock will be reinitialized on the meta data of the disk system and file system.

As our superblock won't change across the life cycle of a session, except the time we initialize the file system, we only need to initialize and write back the superblock once. The user information and bitmap will be deprecated when writing back.

3.2.2 how to resolve the path

The paths have 2 types: absolute path and relative path. The difference when processing them is the start point of the path, which is the root directory for absolute path, and the current directory for relative path. I save the root directory in the superblock, and a curDir variable to store the current directory. As I was mentioned above, the directories are also files, and their names are stored in the iNode. We just need to traverse all links in the Directory inode, check if the destination is directory or not, and move the pointer to the next level directory.

Handling cd commands is just a matter of changing the curDir variable to the destination directory. We pass the string of path to the above function, and the function will return the iNode of the destination directory. If error occurs, the function will return E_ERROR to indicate errors throw error.

3.2.3 How to remove a file or directory

The deletion of a file or directory is removing the iNode, as well as its dependencies to other files or directories, and wipe it out from the disk system.

When we need to delete a directory, we need to check if the directory has some files or not. If it has, we throw an error, otherwise, we can delete the directory recursively, and free each iNode from disk. The deletion of a file is similar, we check if the permissions are valid, and then we can delete it.

3.2.4 How to write or delete data into a file

In the above iNode section, I have mentioned that there are readi() and writei() functions to read or write data into a file. based on the 2 functions, we implement the commands to modify the file.

3.3 support multiple users

In this part, I will implement the support for multiple users. The implementation is based on the iNode system, and the file system design.

In the design, we identify the users by their UID, which is a int. User 1 is the root user. The permissions of file and directory are determined by if the user is the onwer of the file, and the permissions code of the file. I compact the permissions checking functions into a single function. Each time a commands that needs to check the permissions is called, we just need to call the function.

All users are stored in a table in the superblock, as well as their current working directory, if the user CWD is NULL, we set it to the home.

In the design, the permissions bits are divided into 2 parts. The first 3 bits are the owner permissions, the second 3 bits are the other users' permissions. The permissions are represented by 3 bits, which are read, write and execute. The permissions will be checked before any operations.

3.4 Interact with Disk server via TCP socket

In the first part, A TCP server that simulates a disk system is implemented. As the design of my block management priciple is to perform all operations on read_block() and write_block() functions, I just need to fine tuning the 2 functions to make sure they can send and receive messages correctly with disk server.

At the same time, The handling functions in disk server side are also modified to meet the requirements of the file system.
Before a user can access the file system, the file system changes the current working directory and current user to the user specified in the command line. Each time a new user is logged in, it's CWD is set to HOME.

4 Extra features: support multiple concurrent clients

In this part, I will implement the support for multiple concurrent clients. As I have implemented the feature of multiple users, the implementation of multiple clients is just a matter of handling multithreading Programing.

First, As different accesses to the file system kernel may inference each other and cause conflicts, a lock is needed to protect the file system kernel. This is similar to the READER/WRITER problem. I classified the commands operations into 2 types:

- reader operation: which will not make any modification to the file system, such as ls, cd, mkdir, cat;
- writer operation: which will modify some contents within the file system;

Then, the reader-preferred solution is implemented in each handler function, which is to used to protect the file system kernel.

Whats more, the server side should Handle the client connections and disconnections. Ths work need to be protected by another server lock as well, to prevent the server from being crashed by multiple clients. This is called severLock in my implementation, which is to handle the global configurations of client registeration problems.

5 Extra features: Journaling

The kernel of modification to a file is the writei() function. If the server crashes during the modification, We should at least record the modification that has been done, as well the the file size

information. To achieve this, I consider the modification towards a block within an iNode is a transaction, a modification of a block must be atomic, after that, the file size can get updated.

Each time a writing operation is called upon a block, the file size variable will get updated at and only at a successful block writing is done. In that way, the reasonable file size and disk states can be guaranteed.

6 Run and Test the File system

The execution command is as the same as the Project 3 requests pdf.
Run the File system server:
./FS <DiskServerAddress> <BDSPort=10356> <FSPort=12356>
Run the File system client:
./FC <Server Addr> <BDSPort=12356>

Before running the file system, we need to run the disk server first. the command is :
Run the Disk server:
./BDS <DiskFileName> <#cylinders> <#sector per cylinder> <track-to-track delay> <port=10356>
The disk server will run on localhost, and the file system server will run on localhost as well. They differ each other by the port number. The disk server will listen on port 10356, and the file system server will listen on port 12356 , by default.

7 Test the File system

Because the user management structures, the final file system implementation cannot pass the minitest smoothly due To the double log in problem in test codes. So I specified a version that simplify the user management system, and the file system can pass the minitest smoothly. This edition is in ForTest branch

use: **git switch ForTest**

The system is tested by running minitest, as shown in figure, it can pass all the tests.

```
./test_fs
mintest: test:
mintest: block_tests:
mintest:     Running test_read_write_block..
mintest:     Running test_zero_block..
mintest:     Running test_allocate_block..
mintest:     Running test_allocate_block_all..
test_fs: allocate block: No free blocks: Success
mintest:     Running test_free_block..
mintest: inode_tests:
mintest:     Running test_iget..
mintest:     Running test_ialloc..
mintest:     Running test_iupdate..
mintest:     Running test_writei..
mintest:     Running test_readi..
mintest:     Running test_read_write_mixed..
mintest:     Running test_random_binary_read_write..
mintest: fs_tests:
mintest:     Running test_cmd_ls..
mintest:     Running test_cmd_mk..
mintest:     Running test_cmd_mk_invalid..
mintest:     Running test_cmd_mkdir..
mintest:     Running test_cmd_mkdir_invalid..
mintest:     Running test_cmd_cd_absolute..
mintest:     Running test_cmd_cd_relative..
mintest:     Running test_cmd_rm..
mintest:     Running test_cmd_rmdir_with_files..
mintest:     Running test_file_lifecycle..
mintest:     Running test_small_file_ops..
mintest:     Running test_folder_tree_operations..
mintest:     Running test_folder_tree_with_rm..
mintest: 25 passed, 0 failed, 25 total
```

Figure 1: Test the file system

```
./test_bd
mintest: all_tests:
mintest: disk_tests:
mintest:     Running test_cmd_i..
mintest:     Running test_cmd_wr..
mintest:     Running test_wwr..
mintest:     Running test_w_partial..
mintest:     Running test_non_ascii..
mintest:     Running test_out_of_bounds..
mintest: 6 passed, 0 failed, 6 total
```

Figure 2: Test the file system

8 Summary

In this report, I have described the design and implementation of the file system, as well as the evaluations and tests.

The project is implemented in C, and the code is organized into several files, compiled by makefile. The file System that integated with disk server and FS server/client is on the main branch, and the test version is on the ForTest branch. The whole project can be found in the git repository: https://github.com/xiuyunMarx/OS_Project3

Thank for the help of the TA and the professor, I have learned a lot from this project. The project is a good opportunity to practice the knowledge I have learned in the class, and to learn some new things. The project is also a good opportunity to practice the programming skills.