

OS Project3 Report

521030910229 Hao Chiyu

May 26, 2023

Contents

1	Introduction	2
2	Step1	2
2.1	Implementation	2
2.2	Test	3
3	Step2	4
3.1	Implementation	4
3.1.1	Block structure	4
3.1.2	Helper structure	4
3.1.3	Functions	5
3.1.4	The process of program running	5
3.2	Test	6
4	Step3	9
4.1	Implementation	10
4.1.1	Disk	10
4.1.2	File System	10
4.1.3	Client	10
4.1.4	Exit and restart of system	10
4.2	Test	11
4.2.1	Setup	11
4.2.2	f & mk & mkdir & ls	11
4.2.3	cd	12
4.2.4	rm & rmdir	12
4.2.5	cat & w & i & d & e	13
4.2.6	Client reconnect	14
4.2.7	Persistence	14
5	Achievement	15

1 Introduction

The file system is a fundamental component of the operating system. In this project, we have implemented a disk simulator and tiny file system, and connected them together through socket, allowing them to collaborate on processing requests from clients.

2 Step1

In this step, we implemented the disk simulator. I think reading before writing is undefined behavior, so the result of program may be different from demo: "R 2 3" before "W 2 3" will print "NO". :-)

2.1 Implementation

The core of the program is to read and modify storage file, and the entire program structure is shown in Figure 1.

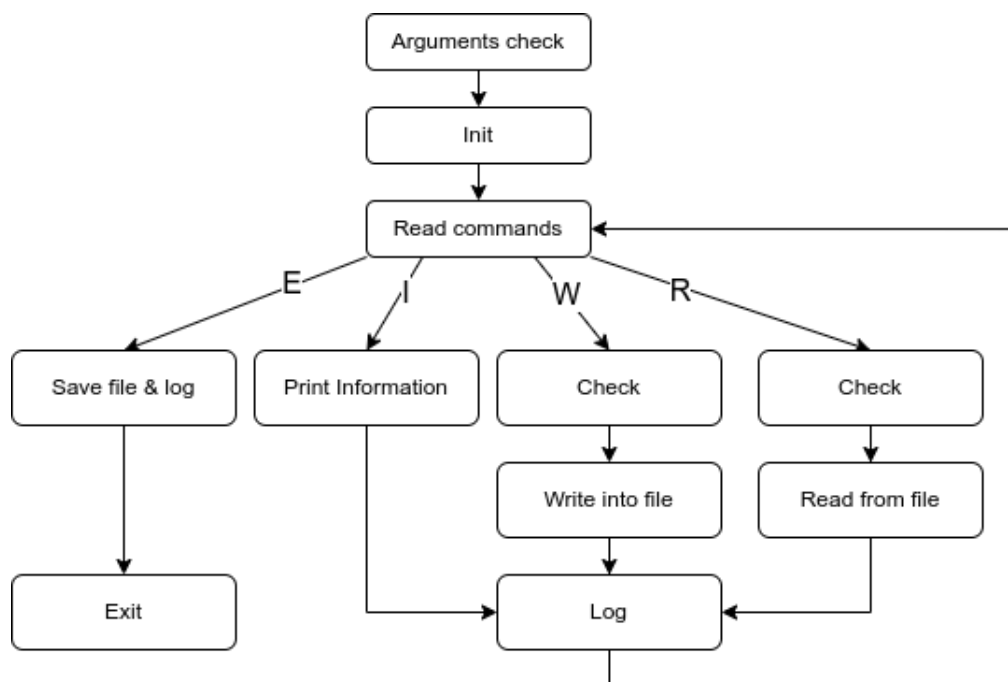


Figure 1: Structure of program of Step 1.

At the beginning of running, the program checks whether the arguments are valid. Then If the arguments format is correct, it will initialize the system according to the arguments, otherwise the program will print an error message and exit.

In the initialization, it sets **CYLINDER_NUM** and **SECTORS_PER_CYLINDER**. Then the program will search whether the disk file name indicated by the arguments already exists in the current directory. If so, the program will set **file_exist** to true and traverse this file to set **cur_max_location**, this variable will be used for command checking, and we will introduce it later.

After initialization, then program will enter a loop, constantly reading commands from the user and executing corresponding functions. Of course, we need to check if the command is valid. By parsing it as **cur_argc** and **cur_argv**, the command is only valid when the length of **cur_argv[0][0]** is 1 and the specified character, otherwise the program will print an error message and wait for the next command. Next, we will introduce the processing methods of each command.

I: I command write **CYLINDER_NUM** and **SECTORS_PER_CYLINDER** into log simply.

R: Firstly, if the stored file does not yet exist, that is, if `file__exist` is false, the program will print an error message because we have not made any writes yet, and this read is illegal. Then the program checks the arguments and parses them. The last check is check if the index of sector we access is greater than `cur__max_location`, because the file pointer can only reach the end of the file at this point , and such reading is an undefined behavior so we let the program to refuse this access. After all checks passed, we can use `fseek()` function to set file pointer and read the content of sector.

W: Firstly, we need to check if `file__exist` is true, which affects the way the file is opened. If the file does not exist, we use `w+` mode to create the file and write it, otherwise we use `r+` mode to write it. This is because using `r+` mode when the file does not exist will cause exceptions, while using `w+` mode when the file exists will empty the existing content in the file. After opening the file in `w+` mode for the first time, set `file__exist` to true. Then we conduct necessary bound check and update `cur__max_location`. Finally, concatenate and write all the parameters after `cur__argv[2]`.

E: This command write "Goodbye!" into log, close log file and make program exit.

To simulate track delay, we declare a variable `cur__cylinder`, every time we access a sector, we get the cylinder index we want to access, then the program wait for microseconds , which is equal to the minus between two cylinder index.

2.2 Test

We test the program according to type script.

```
~/desktop/courses/os/lab/project3/step1 c master !4 c ./disk 4 8 1 disk_storage
I
W 1 2 Hello. world!
R 1 2
YES: Hello. world!
E
~/desktop/courses/os/lab/project3/step1 c master !4 ?1 c
```

Figure 2: First run.

```
1 4 8
2 YES
3 YES: Hello. world!
4 Goodbye!
```

Figure 3: First run log.

```
~/desktop/courses/os/lab/project3/step1 c master !4 ?1 c ./disk 4 8 1 disk_storage
R 1 2
YES: Hello, world!
W 1 2 test test
R 1 2
YES: test test
E
```

Figure 4: Second run.

```

1 YES: Hello, world!
2 YES
3 YES: test test
4 Goodbye!

```

Figure 5: Second run log.

Then we show sample error message in Figure 6 & 7, you can find more types of error in type script.

```

~/Desktop/courses/os/lab/project3/step1 c master !4 ?1 c ./disk 4 8 1 disk_storage
R 0 2
Error: invalid request.
E

```

Figure 6: Example error (access sector have not been written yet).

```

1 NO
2 Goodbye!

```

Figure 7: Example error log.

3 Step2

In this step, we implemented a file system based on inode.

3.1 Implementation

3.1.1 Block structure

Different sectors may have different functions. Depending on their functions, we use different structures to describe this sector and ensure that the size of each structure does not exceed 256 bytes. Next, we will introduce every structure we use in program.

super_block_struct: Super blocks are the foundation of a file system, which contains necessary information for the operation of the file system, such as the starting position of inode bitmap blocks, the number of inode blocks, etc.

inode_bitmap_block_struct: The inode bitmap marks whether an inode has been used. If a bit is true, the corresponding position inode has been used.

data_bitmap_block_struct: Similiar to inode bitmap, the data bitmap marks whether an block has been used. If a bit is true, the corresponding position data block has been used.

dir_block_struct: Used by a directory to save the parent directory, number of child files, name and location of child files.

inode_struct: Used by a file or directory to save the file type, name, direct block index, indirect block index and last update time. It is saved in inode blocks.

inode_block_struct: Save inodes.

indirect_block_struct: Save more block index for a file.

Finally, we declare a union called **Sector_union**, based on all the structures above. We can control the size of **Sector_union** is 256 bytes.

3.1.2 Helper structure

LUT_struct: save last update time, we have functions **inode_LUT_update()** and **LUT_print()** to update and print time.

ls_file_struct: This structure can facilitate the execution of ls command, which includes file names, types, and **LUT_struct**.

file_ptr: This structure is used to execute commands that modify the content of a file, it saves the inode and char pointers of the current file. It can be moved between different data blocks of the file and pointed to the next character through function **file_ptr_next()**.

3.1.3 Functions

is_nature_number(): determine whether a string is a natural number in string format.

stou(): convert natural numbers in string format to integers.

split(): split strings according to specified delimiters, returns the number of substrings and a set of pointers to substrings.

make_ls_struct(): receives a pointer to an inode and generate the corresponding ls_file_struct.

ls_sort(): sort the ls_file_struct array by the file name in lexicographical order.

make_fs_ptr(): the function receives a file inode pointer and an empty file_ptr pointer, and set the file_ptr to point to the first character of the file.

get_inode(): the function receives the index of a file inode in inode blocks, return a pointer to corresponding inode.

get_dir_data_block(): the function receives a pointer to a dir's inode, return a pointer to corresponding dir's data block.

get_file_data_block(): the function receives the sector index of a file's data block, return a pointer to corresponding sector.

bitmap_check(): this function is used to check if there are required number of free bits in the bitmap. This function receive a pointer to a bitmap, required number of free bits and a empty array. If the number of free bits holds the demand, function returns true and puts the index of free bits into array, or it will return false.

get_file_size(): the function receives a file inode pointer, return the size of data blocks which have been allocated to the file by traversing the inode's direct and indirect pointer.

get_content_size(): the function receives a file inode pointer, return the size of the content in the file by set a file_ptr and move it to the end of file's content.

file_resize(): the function receives a file inode pointer and the size wanted, the function get the file size first, then if file size is smaller than wanted size, expand the file by allocating more data block to the file, or shrink the file by delete direct or indirect pointers. Don't forget to modify the bitmaps.

file_delete(): the function receives target file name, then search the name in current directory. If file found, use **file_resize()** free all the data blocks allocated. Finally, we delete the inode in inode block, set inode bitmap and modify the dir data block.

dir_delete(): Similar to **file_delete()**, but we must delete all things in the target dir, so we use **file_delete()** and **dir_delete()** recursively. The last step is the same as **file_delete()**.

3.1.4 The process of program running

First step is initialization, we fix the **CYLINDER_NUM** and **SECTORS_PER_CYLINDER** in this step, so we can directly create simulated disk array **file_storage** and log file. Then we begin to read command.

After reading command, we parse the command with **split()** to set **cur_argc** and **cur_argv**. Of course, we need to do some preliminary checks. For example, the first command cannot be neither "f" nor "e". Next, we will introduce the process of every command.

f: this command will initialize the file system. Firstly, it will set the super block according to the given information. Then, inode bitmap and data bitmap are set to all false, and then basic blocks such as super block's and bitmaps's data block index are set to true. The third step is set inode block, such as set each inode's direct and indirect pointer to -1. At last, we initialize the root directory's inode and data block.

mk: this command will do some checks, including the validity of name, if there is already a file with the same name in current directory, if bitmaps or current dir's data block is full. If all checks passed, a inode and a data block will be allocated to the file. Then this file will be added into current directory.

mkdir: Similiar to "mk", the only one difference between them is that the element "type" in inode will be set as 1, while "mk" sets it as 0.

rm: This command just calls function `file_delete()` and receives its return value. If return value is ture, it means deleting the file successfully, or the program will print error message that the file does not exist.

rmdir: Similiar to "rm", the difference between them is that "rmdir" will look up its children and recursively delete all its children. After its content is cleared, the remain procedure is the same as "rm": free data block and inode, update its parent, etc.

cd: this command receives a path, which is absolute or relative, we use `split()` to split the path into dirs. If the first character of path is '/', which means this path is absolute, we must return to root directory to search dirs. So we set `cur_dir_location` as 0, which indicates the inode index of current directory. Then we can just search the directory iteratively and set `cur_dir_location`.

ls: this command traverses all childs in current directory and generate `ls_file_struct`, then we put these things into `file_temp` and `dir_temp` respectively. At last, we use `ls_sort()` to sort two arraies by the file name in lexicographical order and print.

cat: this command create a `file_ptr` which points to the beginning of file's content, then we can continuously print the character the pointer pointing and call `file_ptr_next()` . When the pointer reach the end of file, `file_ptr_next()` will return false and we can exit from this command.

w: after arguments check, we need to find the file. Then we just use `file_resize()` clear the old content. The third step is combining all the data in command, allocating enough space with `file_resize()`, and write data into file.

i: after arguments check and finding the file. We combine the data we want to write. If the sum of the length of data and the content of file, we call `file_resize()` to allocate new space. If the offset argument is greater than current file size, we can just move `file_ptr` to the end of file and wirte data, or we will declare a temporary array in memory to save the content behind the offset . After writing the data from offset, we can write the content in temporary array back.

d: after necessary checks, if offset is greater than the size of file content, there is no need to delete any thing. If offset plus length are greater than the size of file content we can just delete all character after offset, or we need set another `file_ptr` which points to offset plus length and wirte the content from offset plus length to offset.

e: this command just closes the log file and shutdowns the program.

3.2 Test

We test the program according to type script.

```
~/Desktop/courses/os/lab/project1/step2 ◀ master !3 ?3 ◀ ./fs
> f
> mk a
> mk b
> mk c
> mkdir hello
> mkdir world
> ls
file a 0 Last Update Time: 2023/5/25 21:51
file b 0 Last Update Time: 2023/5/25 21:51
file c 0 Last Update Time: 2023/5/25 21:51
dir hello 256 Last Update Time: 2023/5/25 21:51
dir world 256 Last Update Time: 2023/5/25 21:51
```

Figure 8: Test of f, mk, mkdir and ls.

```
YES: file system inited
YES: create file a
YES: create file b
YES: create file c
YES: create directory hello
YES: create directory world
file  a    0    Last Update Time: 2023/5/25 21:51
file  b    0    Last Update Time: 2023/5/25 21:51
file  c    0    Last Update Time: 2023/5/25 21:51
dir   hello 256  Last Update Time: 2023/5/25 21:51
dir   world 256  Last Update Time: 2023/5/25 21:51
```

Figure 9: Log of f, mk, mkdir and ls test.

```
> cd hello
> mk cat
> cd /world
> mk dog
> cd cd ../../hello
Error: invalid argument.
> cd ../../hello
> ls
file  cat    0    Last Update Time: 2023/5/25 21:51
> cd ../hello/./world
> ls
file  dog    0    Last Update Time: 2023/5/25 21:51
```

Figure 10: Test of cd.

```
YES
YES: create file cat
YES
YES: create file dog
YES
file  cat    0    Last Update Time: 2023/5/25 21:51
YES
file  dog    0    Last Update Time: 2023/5/25 21:51
```

Figure 11: Log of cd test.

```

> cd /
> ls
file a 0 Last Update Time: 2023/5/25 21:51
file b 0 Last Update Time: 2023/5/25 21:51
file c 0 Last Update Time: 2023/5/25 21:51
dir hello 256 Last Update Time: 2023/5/25 21:51
dir world 256 Last Update Time: 2023/5/25 21:51
> rm a
> rmdir hello
> ls
file b 0 Last Update Time: 2023/5/25 21:51
file c 0 Last Update Time: 2023/5/25 21:51
dir world 256 Last Update Time: 2023/5/25 21:51

```

Figure 12: Test of rm and rmdir.

```

YES
file a 0 Last Update Time: 2023/5/25 21:51
file b 0 Last Update Time: 2023/5/25 21:51
file c 0 Last Update Time: 2023/5/25 21:51
dir hello 256 Last Update Time: 2023/5/25 21:51
dir world 256 Last Update Time: 2023/5/25 21:51
YES: remove file successfully.
YES: remove directory successfully.
file b 0 Last Update Time: 2023/5/25 21:51
file c 0 Last Update Time: 2023/5/25 21:51
dir world 256 Last Update Time: 2023/5/25 21:51

```

Figure 13: Log of rm and rmdir test.

```

> w b 100 hello, world!
> cat b
hello, world!
> d b 2 4
> cat b
he world!
> i b 2 9 something
> cat b
hesomething world!
> i b 200 7 append
> cat b
hesomething world!append
> d b 1 1000
> cat b
h
> e

```

~/Desktop/courses/os/lab/project3/step2 c master !4 73 c

Figure 14: Test of cat, w, i, d and e.


```

YES: write to file successfully.
YES:
hello, world!
YES: delete file content successfully!
YES:
he world!
YES: insert to file successfully.
YES:
hesomething world!
YES: insert to file successfully.
YES:
hesomething world!append
YES: delete file content successfully!
YES:
h
Goodbye!

```

Figure 15: Log of cat, w, i, d and e test.

Then we show sample error message in Figure 16 & 17, you can find more types of error in type script.

```

> rm c
Error: file not found.
> rmdir c
Error: directory not found.
> i c 1 1 a
Error: file not found.
> w c 1 1
Error: file not found.
> d c 1 1
Error: file not found.
> cat c
Error: file not found.
> cd c
Error: not found.

```

Figure 16: Example error (access file which does not exist).

```

16 NO: file not found.
17 NO: directory not found.
18 NO: file not found.
19 NO: file not found.
20 NO: file not found.
21 NO: file not found.
22 NO: not found.

```

Figure 17: Example error log.

4 Step3

In this step, we modify the disk simulator and file system program, implement a client program and connect them by socket.

4.1 Implementation

4.1.1 Disk

Based on Step1, we add a new argument **DiskPort**. When we initialize the program, the program will create a socket and listen to **DiskPort**. After connection established, disk program send arguments **CYLINDER_NUM** and **SECTORS_PER_CYLINDER** to file system to initialize it. Next, If the disk storage file already exists, the program will set exist and send a status code of 1, followed by sending the necessary initialization blocks to the file system. Otherwise, 0 will be sent.

After entering loop, the program will continuously receive command from file system and send data back. When file system exits, the disk program will exit too.

4.1.2 File System

Based on Step2, we add two new arguments **DiskPort** and **FsPort**. After connecting to disk program, the program receives **CYLINDER_NUM** and **SECTORS_PER_CYLINDER** from disk program and create necessary array in memory, which store sectors when they need to be transfer into memory. Then the program receives status code from disk, if code is 1, the system will send several "R" commands to get necessary blocks from disk storage file, or we will need to initialize the system by "f" command.

If file system cannot connect to disk, it will wait for 2s and try again, the max number of trials is 5.

After initialization, the file system creates a socket and listens to **FsPort**, waiting for client connecting. After connection established, the file system receives commands from client and executes them. Unlike Step 2, all access to array in memory by the file system in Step 2 needs send commands to disk program. Such as "i" command, we need to transfer blocks needed to memory, modify them and write back to disk program. So in Step3, every time we modify a block, this block's sector index will be put into a array. At the end of every command, all sectors contained in array will be write back to disk to update disk storage file.

If client exits, the program will find another free port as **FsPort**, waiting for next client connecting.

To implement the functions we need, here are new functions we add:

get_sector_from_disk(): this function receives the index of sector we want to get from disk, then format it as a "R" command, which is sent to disk. The function will receives the data and write into memory. It sets **data_block_in_memory** flags so that we don't need to access disk next time.

transfer(): this function transfers sectors by files, it receives the file inode index and finds all the sectors, which are allocated to the file, calls **get_sector_from_disk()** to get these sectors. It sets **file_in_memory** flags so that we don't need to access disk next time.

sector_write_pre(): this function will add a sector into the queue, the sectors in the queue are needed to write back to disk to update data.

update_disk(): this function will sort and remove duplicated sector in the queue , which is introduced above. Then the function will format the indexes of sectors as "W" commands, send them back to disk to update data.

4.1.3 Client

The client program is relatively simple. It just try to connect to **FsPort**. Then it gets command from stdin and send it to file system, receives output from file system and print. If client cannot connect to file system, it will wait for 2s and try again, the max number of trials is 5.

Before the client exiting, it will send a special command to inform file system its exiting.

4.1.4 Exit and restart of system

In this step, we implement graceful shutdown for disk and file system program. We have implement a function **m_kbhit()**, at the every loop in file program, the function can check if there are any input from file system's stdin nonblockly. The program will check if the character input is 'q', if so, a variable **file_system_exit_signal** will be set as true. The program will wait for the exiting of client, but still handle the commands from client. After exiting, the file system will write basic blocks into disk, send

exit command to disk, close socket and log file pointer and exit. Thus we can sustain the persistence of system's data.

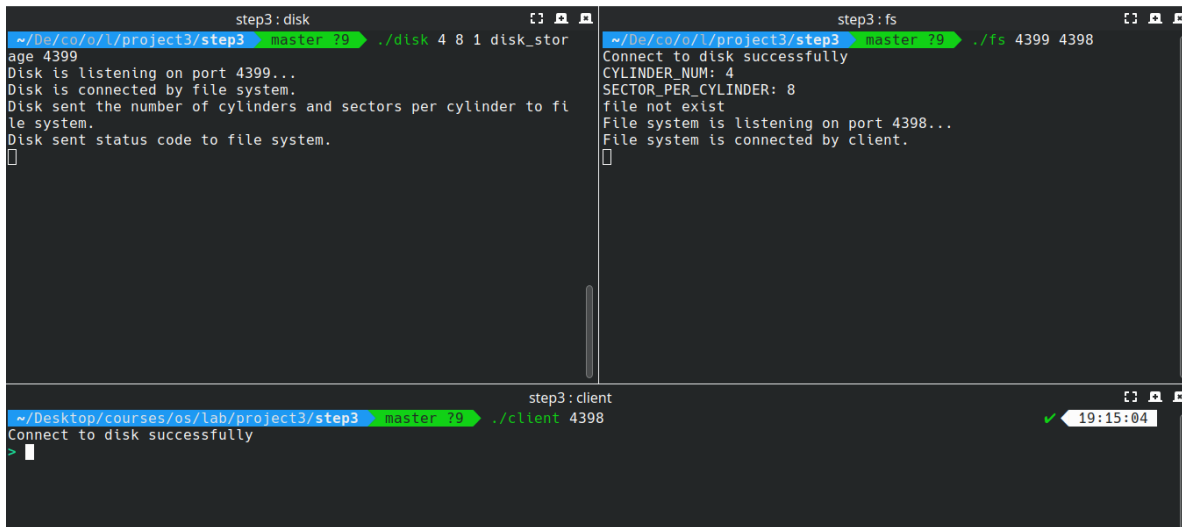
If system restarted, disk program will find the disk storage file in the last running, and send information to file system, thus we can read and modify the data we handle before.

4.2 Test

We test the program according to type script, which is approximately the same to Step2.

4.2.1 Setup

Run disk program first, then set up file system program and connects to disk. At last, run client program.



```
step3: disk
~/Desktop/courses/os/lab/project3/step3 master 79 ./disk 4 8 1 disk_stor
age 4399
Disk is listening on port 4399...
Disk is connected by file system.
Disk sent the number of cylinders and sectors per cylinder to file system.
Disk sent status code to file system.
█

step3: fs
~/Desktop/courses/os/lab/project3/step3 master 79 ./fs 4399 4398
Connect to disk successfully
CYLINDER_NUM: 4
SECTOR_PER_CYLINDER: 8
file not exist
File system is listening on port 4398...
File system is connected by client.
█

step3: client
~/Desktop/courses/os/lab/project3/step3 master 79 ./client 4398
Connect to disk successfully
> █
```

Figure 18: Set up all programs.

4.2.2 f & mk & mkdir & ls

In Figure 19, we use "f" command to initialize the file system, then we create 3 files and 2 directories with "mk" and "mkdir" commands. At last, "ls" helps us view files and dirs in the current directory.

```

> f
Success: file system initialized.
> mk a
> mk b
> mk c
> mkdir hello
> mkdir world
> ls
file  a    0    Last Update Time: 2023/5/25 21:58
file  b    0    Last Update Time: 2023/5/25 21:58
file  c    0    Last Update Time: 2023/5/25 21:58
dir   hello 256   Last Update Time: 2023/5/25 21:58
dir   world 256   Last Update Time: 2023/5/25 21:58

```

Figure 19: Outputs of f, mk, mkdir and ls commands.

4.2.3 cd

Figure 20 show the output of command "cd". This time, we will respectively enter the "hello" and "world" dirs and create "cat" and "dog" files in these two subdirectories. The first and third "cd" commands use relative paths, while the rest use absolute paths. At the same time, we can also find that programs can handle "." and "..".

```

> cd hello
> mk cat
> cd /world
> mk dog
> cd ../../hello
> ls
file  cat    0    Last Update Time: 2023/5/25 21:58
> cd ../hello/../../world
> ls
file  dog    0    Last Update Time: 2023/5/25 21:58

```

Figure 20: Usage of cd.

4.2.4 rm & rmdir

In figure 21, we use "rm" and "rmdir" commands delete file "a" and dir "hello".

```

> cd /
> ls
file  a    0    Last Update Time: 2023/5/25 21:58
file  b    0    Last Update Time: 2023/5/25 21:58
file  c    0    Last Update Time: 2023/5/25 21:58
dir   hello 256   Last Update Time: 2023/5/25 21:58
dir   world 256   Last Update Time: 2023/5/25 21:58
> rm a
> rmdir hello
> ls
file  b    0    Last Update Time: 2023/5/25 21:58
file  c    0    Last Update Time: 2023/5/25 21:58
dir   world 256   Last Update Time: 2023/5/25 21:58

```

Figure 21: Usage of rm and rmdir.

4.2.5 cat & w & i & d & e

In Figure 22, we use command "i", "w" and "d" modify the content of file, and use "cat" view the content. At last, we use "e" to shutdown the program.

```
> w b 100 hello, world!
> cat b
hello, world!
> d b 2 4
> cat b
he world!
> i b 2 9 something
> cat b
hesomething world!
> i b 100 7 append
> cat b
hesomething world!append
> d b 1 100
> cat b
h
> e
Exit
```

Figure 22: Usage of cat & w & i & d & e.

4.2.6 Client reconnect

In Figure 23, we exit client program and restart it, this time the file system program listens to port 4401. After connection established, we write some words into file "b", which is created in previous run, then we exit again. At last connection, we read "b" and the program print the content we write before.

```
~/De/co/o/l/project3/step3 fix !1 ?9 ./client 4401
Connect to disk successfully
> i b 0 persistence test
> e
Exit

~/De/co/o/l/project3/step3 fix !1 ?9 ./client 4402
Connect to disk successfully
> cat b
persistence test
> e
Exit
```

Figure 23: Reconnect: client view point.

```
File system is listening on port 4401...
File system is connected by client.
Command: i b 0 persistence test
Command: exit
Client exit.
Retry port 4402
File system is listening on port 4402...
File system is connected by client.
Command: cat b
persistence test
Command: exit
Client exit.
Retry port 4403
File system is listening on port 4403...
```

Figure 24: Reconnect: file system view point.

4.2.7 Persistence

After last test, we reconnect and type "q" in file system, it will exit after client exits. Then we restrat all 3 programs and read b, the program will print content we write before.

```
~/De/co/o/lab/project3/step3 fix !1 ?9 ./client 4403
Connect to disk successfully
> cat b
persistence test
> □
```

Figure 25: Persistence test: client view point.

```

File system is listening on port 4403...
File system is connected by client.
q
Command: exit
Waiting for client exiting...

~/De/co/o/l/project3/step3 fix !1 ?9 ./fs 4399 4403
Connect to disk successfully
CYLINDER_NUM: 4
SECTOR_PER_CYLINDER: 8
file exist
File system is listening on port 4403...
File system is connected by client.
Command: cat b

```

Figure 26: Persistence: file system view point.

Then we show sample error message in Figure 27 & 28, you can find more types of error in type script.

```

~/De/co/o/l/project3/step3 fix !1 ?9 ./fs 4380 4398
Retry...
Retry...
Retry...
Retry...
Connect failed, please set up disk.

```

Figure 27: Sample Error(faill to connect server).

```

> rm c
Error: file not found.
> rmdir c
Error: dir not found.
> i c 1 1
Error: file not found.
> w c 1 1
Error: file not found.
> d c 1 1
Error: file not found.
> cat c
Error: file not found.
> cd c
Error: dir not found.
>

```

Figure 28: Sample Error(access file or dir which does not exist).

5 Achievement

This project is the largest one I have completed since entering university. Completing the project gave me a great sense of achievement. And this experience has also enhanced my understanding of operating system, especially file system.

Thank you to Prof. Wu and Prof. Chen. Their course 'Modern Operating Systems' taught me the basic concepts of operating systems.

Thank you to Teacher Li. Thank you for bringing me three interesting and challenging projects in the course "Operating System Curriculum Design".

Thank you to the TAs. Thank you for their hard work and guidance.

Thank you to my classmates. Thank you all for spending a semester with me.