# CSC3150 Project4 Report

Liang MingRui 120090723

2022.11.26

## Environment of running my program:

I use the environment in the cluster provided by TA. I also use the computer in TC301 to verify my output.

Brief information:

OS is LINUX

VS code version is 1.73

CUDA version is 11.7.99

A screenshot of information:

**Cluster Environment**

The following information holds for every machine in the cluster.

| Item | Configuration / Version |
|---|---|
| System Type | x86_64 |
| Opearing System | CentOS Linux release 7.5.1804 |
| CPU | Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz 20 Cores, 40 Threads |
| Memory | 100GB RAM |
| GPU | Nvidia Quadro RTX 4000 GPU x 1 |
| CUDA | 11.7 |
| GCC | Red Hat 7.3.1-5 |
| CMake | 3.14.1 |

## Execution steps of running your program:

There is a .sh file provided by TA in the folder. I can directly use command “sbatch slurm.sh” to run my program.

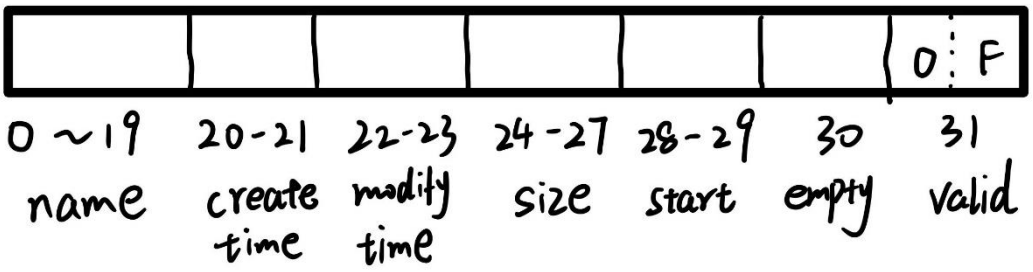
## My way to design my program:

**Source：**

1. *Big picture design*:

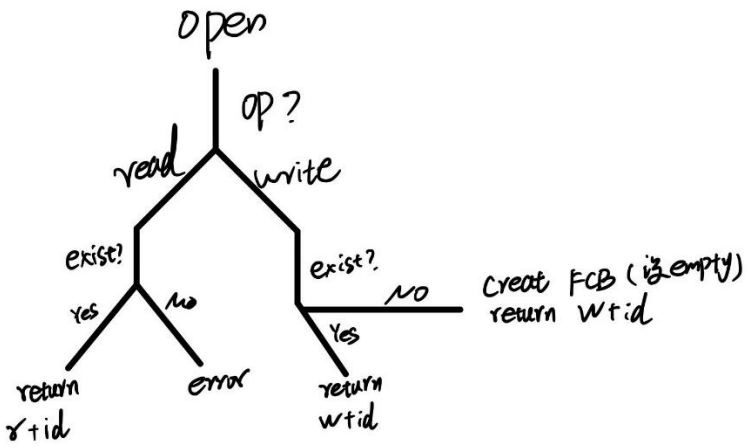
The whole volume contains 1060KB, which is divided into three parts. The first part is super block, which contains 4KB. It is used to store the information that whether the corresponding block in file space is occupied. The second part is FCB, which contains 32KB. It is used to store the information of a file. The last part is file space, which contains 1024KB. It is used to store the content in the file.

For FCB, totally 32B for one entry. The 0~19B is used to store the name, 20~21B is used to store the create time, 22~23B is used to store the modify time, 24~27B is used to store the size of the file, 28~29B is used to store the start block in the file space, 30B is used to store the file is empty or not and 31B is used to store whether the FCB entry is valid or not.

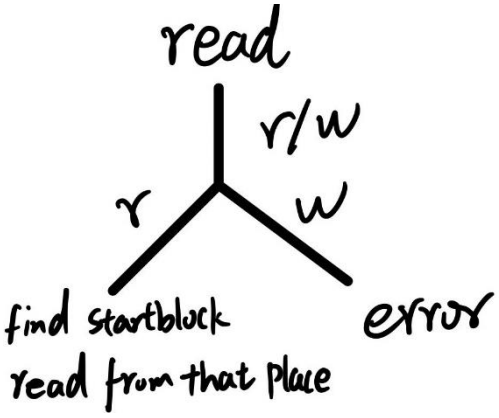



2. *Function design*:

**fs_open**: In this function, I use op to decide if the operation is read or write. If read, then to check if the file exists by name. If yes, then return the fcb_id. If no, the print error. If write, then also to check whether the file exists. If yes, return fcb_id. If no, then find an empty FCB entry for the file and return that fcb_id.



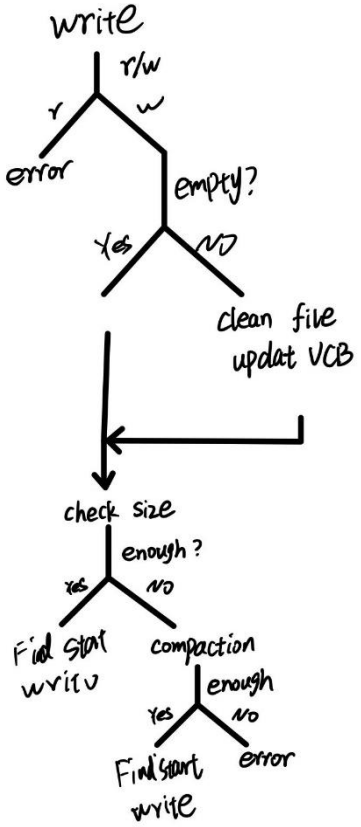

**fs_read**: Firstly, check if the fp is opened for read. If no, then print error. If yes, then find the startblock from fcb and read file from that address.




**fs_write**: Firstly make sure the sp is opened for write. Second, checking the empty byte. If it's not empty, clean the file and update the corresponding vcb. Third step, checking the file size, if the remaining space is not enough, do compacting. Then to check the size again, if the space is still not enough, then print error. Otherwise, finding the startblock from FCB and write content to the file space.

By the way, for the function compact, I need to use the vcb bit to finish

it. I find the first 1 bit and the first 0 bit. And move the content form first 0 bit to first 1 bit through file size. After that, the function needs to update the FCB. The function needs to do this again and again until all the file compact together.




**Fs_gsy(RM)**: Firstly using the filename to search the file's fcb_id in 1024 FCB entry. Then to clean the file content and corresponding vcb bits with the help of information of FCB. After that, clean the FCB rntry.

**fs_gsy(LS_D)**: The function is used to print the file in modify time order. Setting an upper bound to avoid double print. Using a while loop and for loop in 1024 FCB entry to print the filename, every time find the local max modify time, then print it out and antler the upper bound.

**fs_gsy(LS_S)**: The function is used to print the file in size order. And when the sizes are same, using the create time to judge which file should be printed. I also use the while loop to print the filename. Different than LS_S, I create a array to recode if the file has been printed. Using a for loop in 1024 FCB entry to find the max size like LS_S. and if the size are same, then the function will go to compare the create time.

**Bonus:**

1. *Big picture design*:

It is similar to basic case. Only few points are different. One is the valid byte 0x0f means file and 0xff means folder in the FCB entry. And the size in folder FCB entry is the total size of it's files' or subfolders' name-size. Another different point is I add a global variable to recode the current folder's fcb_id. Third different point is all information about it's files and subfolder is stored in the corresponding file space. The file size of folder is statically 128B, which is 4 blocks. The first byte in the file space is treated as file number in the folder. The 2~3B are treated as the parent of the folder. Remaining every 2 byte is treated as its files fcb_id.

In the beginning, I use the first FCB entry to be the root folder.

2. *Function design*:

**fs_open**: This function is similar to the basic case, the only different place is the function will search the current folder instead of the whole volume. Also, the open will affect the modify time of the current folder and

its parent folders.

**fs_read/fs_write**: This two functions can be almost copied from basic case.

**fs_gsy(RM)**: This function only search file in current folder and change modify time.

**fs_gsy(LS_D/LS_S)**: Only difference between basic and bonus is this two function will only search file in current folder in for loop instead of 1024 FCB entry.

**fs_gsy(MKDIR)**: This function will find a new FCB entry to initiate the folder. If the file size is not enough, it will do the compact.

**fs_gsy(CD/CD_P)**: This two function will find the required fcb_id by filename in the file space in current folder.

**fs_gsy(PWD)**: this function will find its parents fcb_id and use the fcb_id to get those names. After that, the function will print them out in certain order.

**fs_gsy(RM_RF)**: this function will find the folder needed to be deleted by filename. It will clean all the content in the folder using for loop and recursion. When doing the deletion, if is a file than delete the file, if is a folder, then into that folder and delete all the content in the folder. After that, the function will update the current folder.

## Problem I met in the assignment:

1. How to store the information in a FCB entry?

*Solution*: Arranging some bytes in right size for different attributes.

2. How to do the compaction?

*Solution*: Using VCB can easily check the empty block and filled block. After that, I can modify the bit in VCB to present that I have change the position of the file content. And I need to update the FCB entry.

3. How to store the information of subfiles and subfolders?

*Solution*: The FCB is obviously not large enough. Thus, I store those information in the file space and arrange their occupied place can also make me very easy to find those information.

4. The number being masked in not equaled to the mask.

Solution: I use mask – (number&mask)==0 instead of mask == number&mask to make this comparison work.

## Screenshot:

**Test Case1:**

```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by file size===
t.txt 32
b.txt 12
===sort by modified time===
b.txt
t.txt
===sort by file size===
b.txt 12
```

**Test Case2:**

```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by file size===
t.txt 32
b.txt 12
===sort by modified time===
b.txt
t.txt
===sort by file size===
b.txt 12
===sort by file size===
*ABCDEFGHIJKLMNOPQR 33
)ABCDEFGHIJKLMNOPQR 32
(ABCDEFGHIJKLMNOPQR 31
'ABCDEFGHIJKLMNOPQR 30
&ABCDEFGHIJKLMNOPQR 29
%ABCDEFGHIJKLMNOPQR 28
$ABCDEFGHIJKLMNOPQR 27
#ABCDEFGHIJKLMNOPQR 26
"ABCDEFGHIJKLMNOPQR 25
!ABCDEFGHIJKLMNOPQR 24
b.txt 12
===sort by modified time===
*ABCDEFGHIJKLMNOPQR
)ABCDEFGHIJKLMNOPQR
(ABCDEFGHIJKLMNOPQR
'ABCDEFGHIJKLMNOPQR
&ABCDEFGHIJKLMNOPQR
b.txt
```

**Test Case3(part of it):**

```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by file size===
t.txt 32
b.txt 12
===sort by modified time===
b.txt
t.txt
===sort by file size===
b.txt 12
===sort by file size===
*ABCDEFGHIJKLMNOPQR 33
)ABCDEFGHIJKLMNOPQR 32
(ABCDEFGHIJKLMNOPQR 31
'ABCDEFGHIJKLMNOPQR 30
&ABCDEFGHIJKLMNOPQR 29
%ABCDEFGHIJKLMNOPQR 28
$ABCDEFGHIJKLMNOPQR 27
#ABCDEFGHIJKLMNOPQR 26
"ABCDEFGHIJKLMNOPQR 25
!ABCDEFGHIJKLMNOPQR 24
b.txt 12
===sort by modified time===
*ABCDEFGHIJKLMNOPQR
)ABCDEFGHIJKLMNOPQR
(ABCDEFGHIJKLMNOPQR
'ABCDEFGHIJKLMNOPQR
&ABCDEFGHIJKLMNOPQR
b.txt
===sort by file size===
~ABCDEFGHIJKLM 1024
}ABCDEFGHIJKLM 1023
|ABCDEFGHIJKLM 1022
```

**Test Case4(part of it)：**

```
triggering gc
===sort by modified time===
1024-block-1023
1024-block-1022
1024-block-1021
1024-block-1020
1024-block-1019
1024-block-1018
1024-block-1017
1024-block-1016
1024-block-1015
1024-block-1014
1024-block-1013
1024-block-1012
1024-block-1011
1024-block-1010
1024-block-1009
1024-block-1008
1024-block-1007
1024-block-1006
1024-block-1005
1024-block-1004
1024-block-1003
1024-block-1002
1024-block-1001
1024-block-1000
1024-block-0999
1024-block-0998
1024-block-0997
1024-block-0996
1024-block-0995
1024-block-0994
1024-block-0993
1024-block-0992
1024-block-0991
1024-block-0990
```

## Bonus Test Case:

```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by modified time===
app d
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
app 0 d
===sort by file size===
===sort by file size===
a.txt 64
b.txt 32
soft 0 d
===sort by modified time===
soft d
b.txt
a.txt
/app/soft
===sort by file size===
B.txt 1024
C.txt 1024
D.txt 1024
A.txt 64
===sort by file size===
a.txt 64
b.txt 32
soft 24 d
/app
```

```
===sort by file size===
t.txt 32
b.txt 32
app 17 d
===sort by file size===
a.txt 64
b.txt 32
===sort by file size===
t.txt 32
b.txt 32
app 12 d
```

## Things I learned in this assignment:

1. I learned how to implement a file system.
2. I learned how to do compaction in a file system.
3. I learned the usage of VCB, FCB and file space.
4. I learned how to do a multilayer folder in a file system.
5. I learned how to record and use different meta data in different cases.