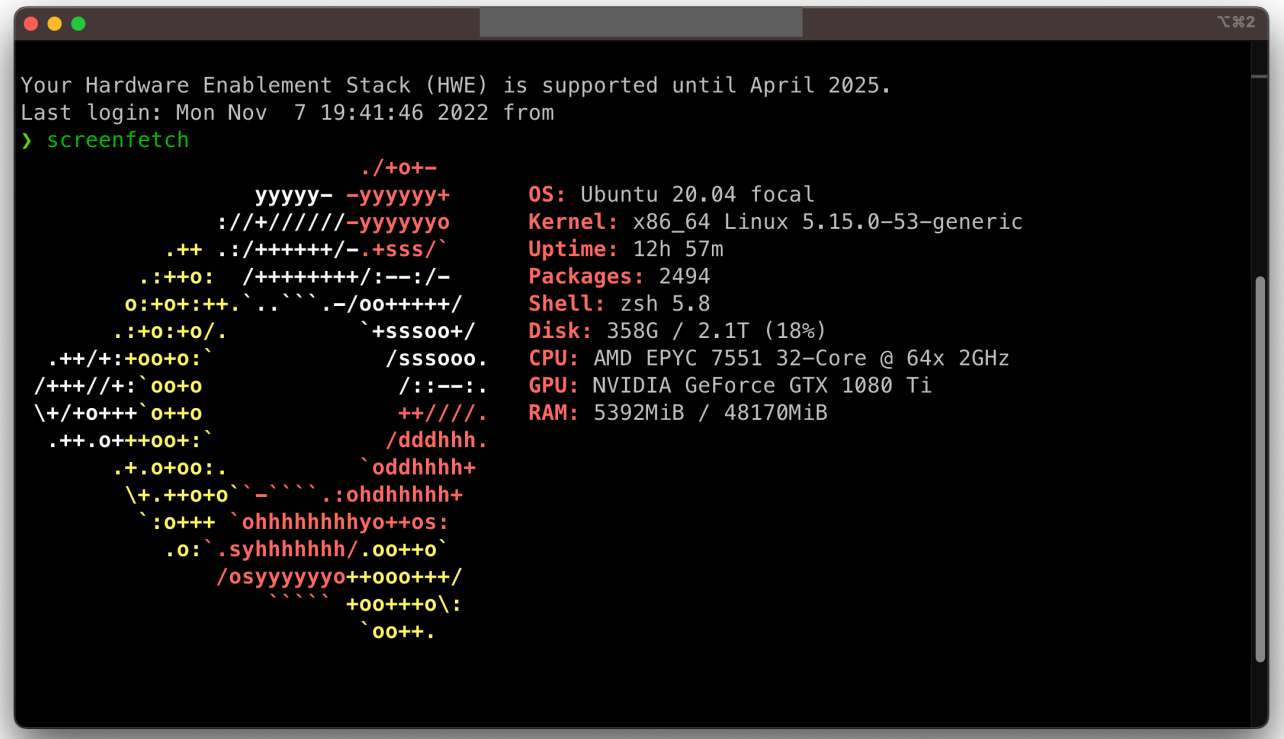
CSC 3150 Assignment 4 Report

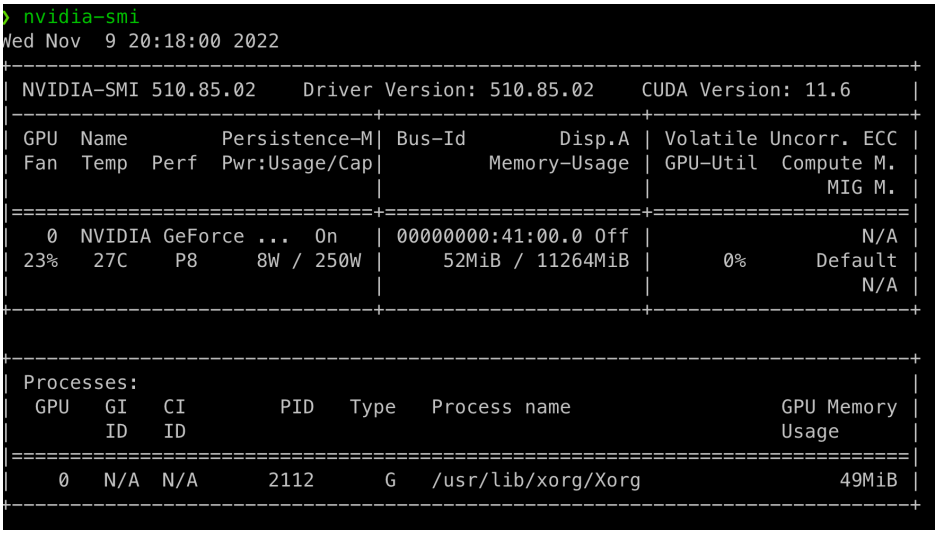
Tianhao Shi, 120090472

Nov, 2022

Environment



CUDA Version: 11.6



Also tested on the provided cluster

Execution

Method 1: Use makefile

```
cd Assignment_3_120090472/source
make #(compile+run)
# make build #(build only)
```

Method 2: Batch script

```
cd Assignment_3_120090472/source
bash ./slurm.sh
```

Overview of Project

This project simulates a basic file system using the GPU memory. There is only one volume, whichi is further divided into three parts:

1. 4KB Volume Control Block
2. 32KB File Control Blocks (32B*1024)
3. 1MB Space for storing the actual component of the file content.

Aside from the main volume, there is also a 128B temporary space for global variables. This file system is created and initialized by launching a single thread *(<<<1, 1>>>)* with an input buffer and a output buffer in the main file (`main.cu`). The input buffer stores all the data to be written to the file system, while all the read operations stores data in the output buffer. The set of operation on the file system is defined in the `user_program()` function within the `user_program.cu` file. The whole file system volume can be viewed as a very large 1-d uchar array, and all the operations are manipulating the array.

Design of key components

Volume Control Block (VCB)

In this file system, each file occupies some bytes. However, to manage files easily, we group every 32 bytes of the file as a block. When allocating space for files, we always allocate entire blocks to files, even if the file size is not a multiple of 32 (which means this file system has **internal fragmentation**). To keep track of the free space on the disk, we use the volume control block. This block serves like a 'snapshot' of the actual storage area.

The underlying data structure of the VCB is a bitmap. In my implementation, each byte of the VCB represents the status of 8 blocks. 1 means the block is occupied, while 0 means the block is available for allocation. Since the VCB area is 4KB, there are 4K*8=32K blocks. These blocks can store 32KB*32KB=1MB data, which corresponds to the size of the storage area for data.

File Control Blocks (FCB)

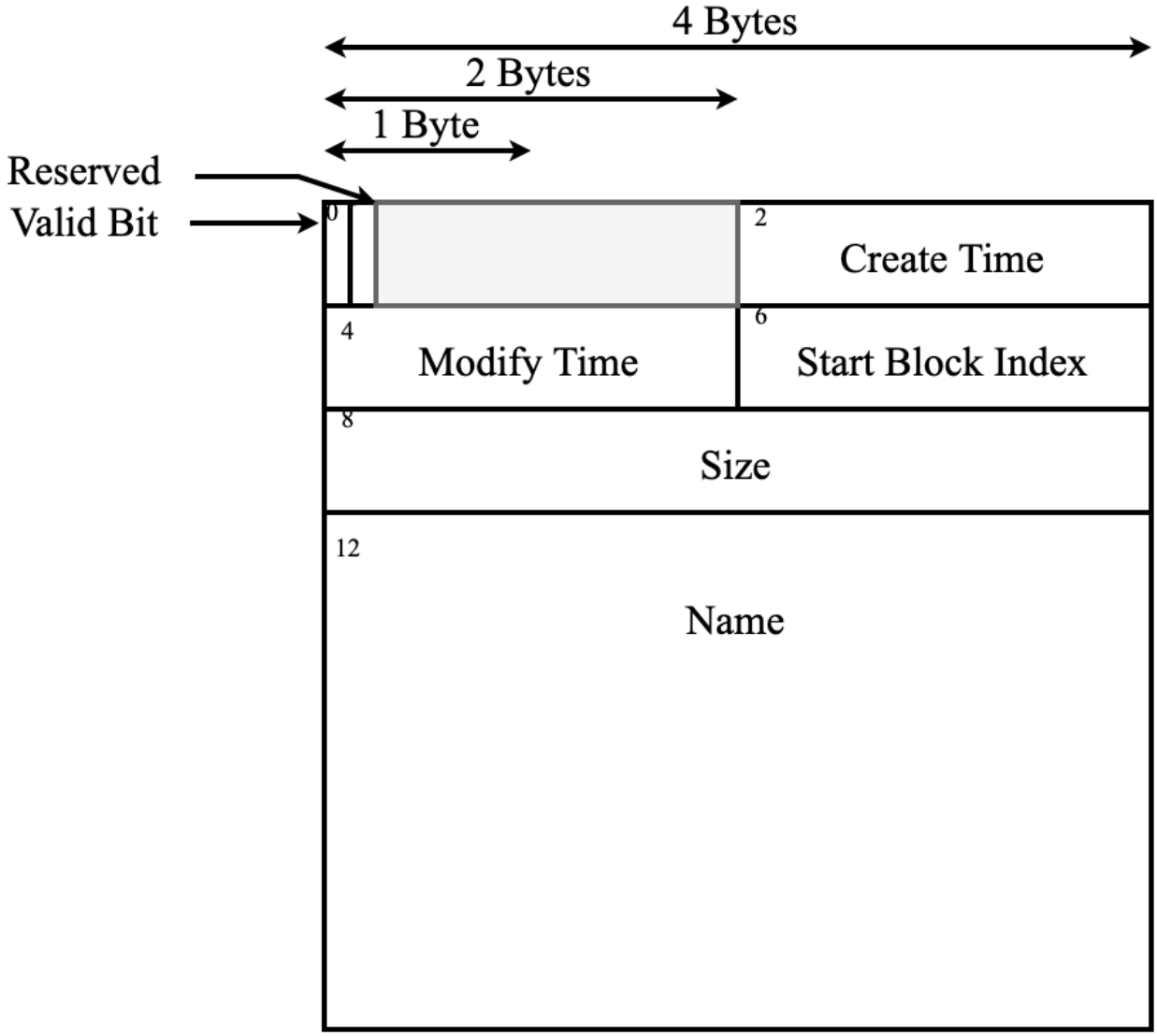
As mentioned above, the VCB only records the availability of the disk space. To identify different files, we also need a place to store auxilary information about files. The information about a file is called its **metadata**. We use 32 Bytes to record various information of a file. Each of these 32 Byte space is called a **File Control Block**. The size of FCB is consistent for every file, and the type of information stored in a particular area within the FCB is also the same for different files.

Source

In the basic part of the project, each FCB stores the following information:

- Valid Bit: 1 bit, indicates whether this FCB is valid. If it's cleared, the rest attributes will not be valid.
- Create Time: 2 byes, indicates the creation time of a file. Set when the file is created and never changed.
- Modify Time: 2 bytes, indicates the most recent update time of the file. Set when: a) the file is created; b) when the file is updated.
- Start Block Index: 2 bytes, indicates the index of the starting block of the file. Using this information, we can acquire the base address of a file. Set whenever the file is write. May change whenever the file is allocated to a different location (to be discussed later in `fs_write()`).
- Size: 4 bytes, indicates the actual size of the file. When the file is created, it is set to 0. Later its value will be set whenever the file is written. I use 4 bytes because the maximum size of the file is 1MB (this is the case when there is only 1 file occupying the whole storage area). Although using 3 bytes is also enough, since 3*8>20, to guarantee address aligness, I still use 4 bytes.
- Name: 20 bytes (may be smaller), indicates the actual name of the file. Note that whenever we set or read the file, we use a char* pointer, which retrives until the first /0.

The following graph illustrates how I use the 32 bytes of the FCB. Note that for the sake of illustration, I showed FCB as a 2D array, but in fact it should be an 1d array.



Bonus

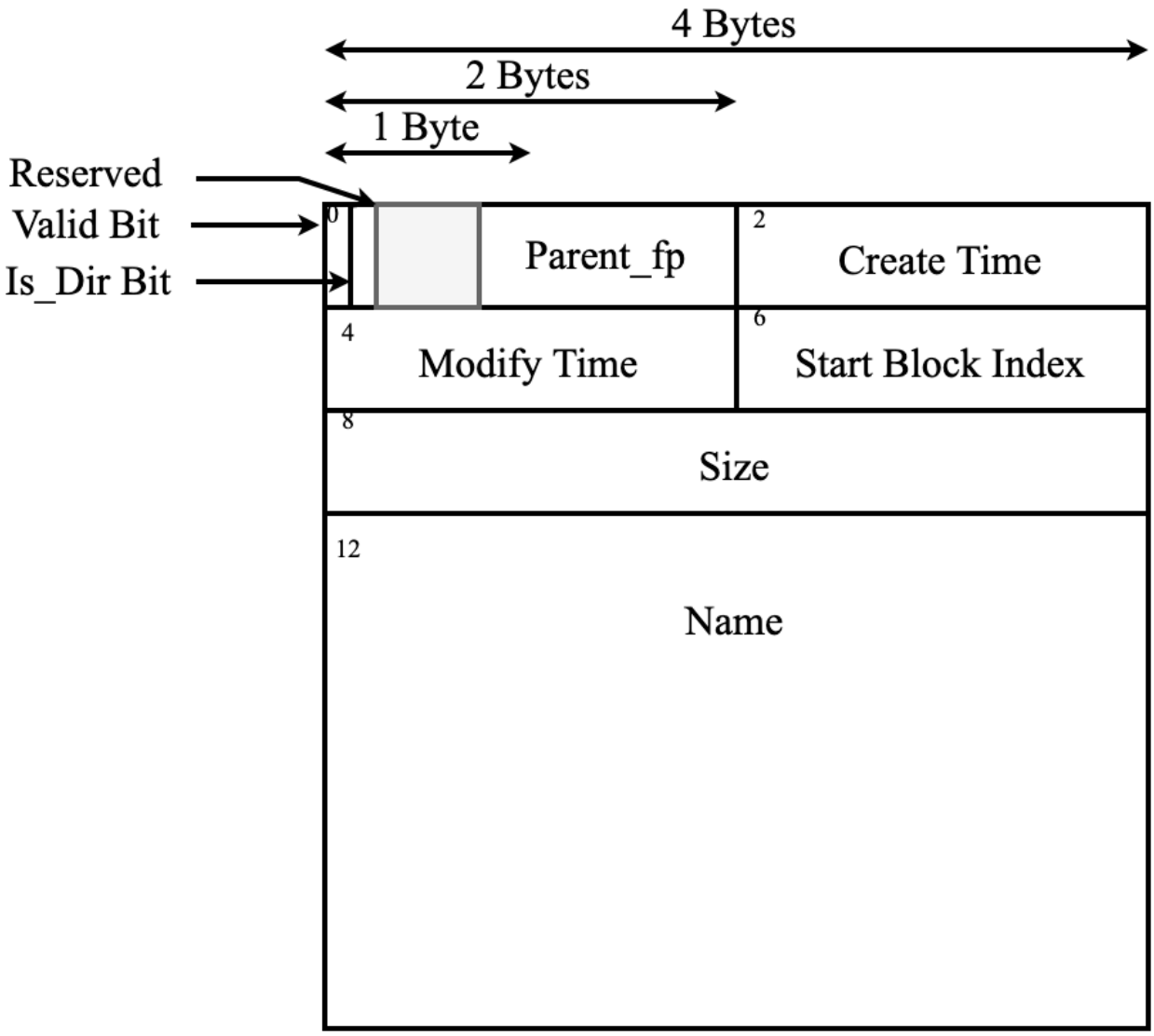
It should be noticed that in the basic version of the project there is no subdirectory, and every file is stored under the root directory. However, in the bonus part we have to design the tree-structure. This indicates there is always 1 FCB used to record the metadata of the root directory (in my case, I use FCB[0]). Moreover, introducing subdirectories also poses the following challenge:

1. How to differentiate a file from a folder(directory)?
2. How to identify the parent of a file/directory?

To tackle these challenges, I introduced two attribues to the FCB:

1. is_dir bit (1 bit): indicates whether this is a file or a folder. 1 means this is a folder, and 0 means this is a file. This bit limits operation allowed on this FCB, as I will discuss later.
2. parent_fp (10 bits): indicates the position of a folder/directory's parent directory. Basically it's the index of that directory's FCB

In the actual operation, these two attributes, together with the valid bit is always retrived together by taking 2 bytes (MISC_ATTR_LENGTH) starting from 0th byte. The actual value of each attribute is acquired through bit operations. The updated FCB is illustraed as follows:



File Pointers

The file pointer contains information about the FCB index of a file. In the user level, i.e, when passed between provided API functions like fs_open()/fs_read()/fs_write(), its least significant bit in binary representation states the type of that file pointer. There are two types of file pointers in the user leve:

1. Read file pointer: LSB is 0
2. Write file pointer: LSB is 1

When passing a file pointer to an incompatable function (e.g, passing a write file pointer to fs_read()) will cause an error.

Aside from the LSB, the other bits are simply the index of the file's FCB among the 1024 FCB entries. This means its value, if valid ranges between 0 and 1023. When it equals to 1024, it indicates the requested file is not present in the FCB table.

When the fp is not passed in the user level, its type is not important anymore, so we ommit the type bit. I.e, when not passed to fs_read()/fs_write(), the file pointer value equals to the index of the file's FCB entry within the FCB table.

Content of the files

The content of the file is stored starting from the base address (4K+32K=36K). Each byte of the file corresponds to a ucar within the volume. The base address, as described earlier, can be acquired using the stary byte:

$$start\_address = storage\_base\_addr + block\_idx \times block\_size$$

and the size of the file can also be acquired from the FCB's metadata.

Temporary Usage

Source

In the basic version of this project, 12/128 bytes of temporary usage space is utilized to store 3 global variables:

- gtime: the current time of the file system. Note that I assume there will be less than 2^{16} operations in the file system. Also, in my design, each command contributes to at most 1 increment in the current time. The details will be discussed in the function design part. In this system, the time is rigorously increasing. On initialization it is set to 0.
- gfilenum: the current number of file within the file system. On initialization, it is set to 0.
- glastblock: the index of next possibly available block. This variable is used because I use **next fit algorithm** to allocate space for new file (or to move old file to a new position). On initialization, it is set to 0, because the 0^{th} block is available.

Bonus

In the bonus part, 20/128 bytes of the temporary space is used. Aside from the 3 global variables, I add two more variables:

- gcwd: tracks the current directory's FCB index. On initialization, it is set to 0, indicating a filesystem is in the root directory.
- glevel: tracks the depth of current directory. the root directory is located in the 0^{th} level. Every file within the root directory is in the 1^{st} level. Therefore, on initialization, this variable is set to 1.

Design of Functions

`fs_init()`

This function is used to initialize the file system. Aside from the code provided in the template, I add the following codes:

source

1. Clear all the bytes in the VCB area.
2. Invalidate all the FCB entries.

bonus

1. Create the root directory's FCB using the `fs_gsys(MKDIR)`
2. It should be noted that the time after `fs_init` should be 0, but creating the root FCB increases the system time. Therefore, we should reset the system time.

`fs_open()`

This function receives a file name and returns a user-level file pointer(LSB indicates mode).

source

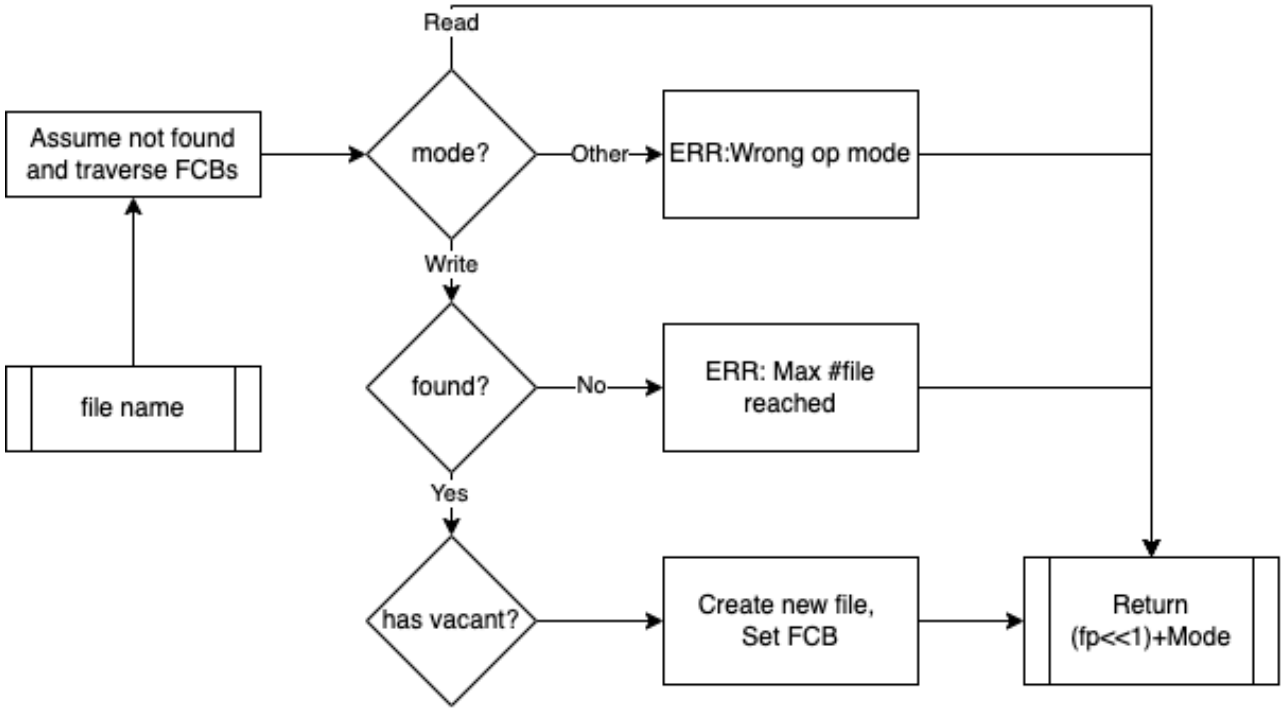
First, the function will traverse the FCB table until:

1. It found a FCB with name same as the requested name
2. It traversed all valid FCB and found a vacant FCB

When the requested filename is not present:

1. If the file is opened in *read* mode, an error is raised
2. If the file is opened in *write* mode
    1. If there is a vacant FCB, it will create a new file. This increases the file number and the system time
        - The *valid bit* is set
        - The *size* is set as 0
        - The *create time* and *modify time* is set as the current system time (before increment)
        - The *name* is set as the requested
        - No block is allocated on creation
    2. Else, it means maximum number of file is reached, and an error is raised

On the other hand, if the filename is present, this functin simply left shifts the found FCB index by 1 bit and append the mode of the file pointer.



bonus

The bonus implementation of `fs_open()` is almost identical with the basic version. Except that when searching for the filenames in the FCBs, we have to make sure the filename and the the parent_fp are both the same. Since a file with same name could appear in different directory.

`fs_write()`

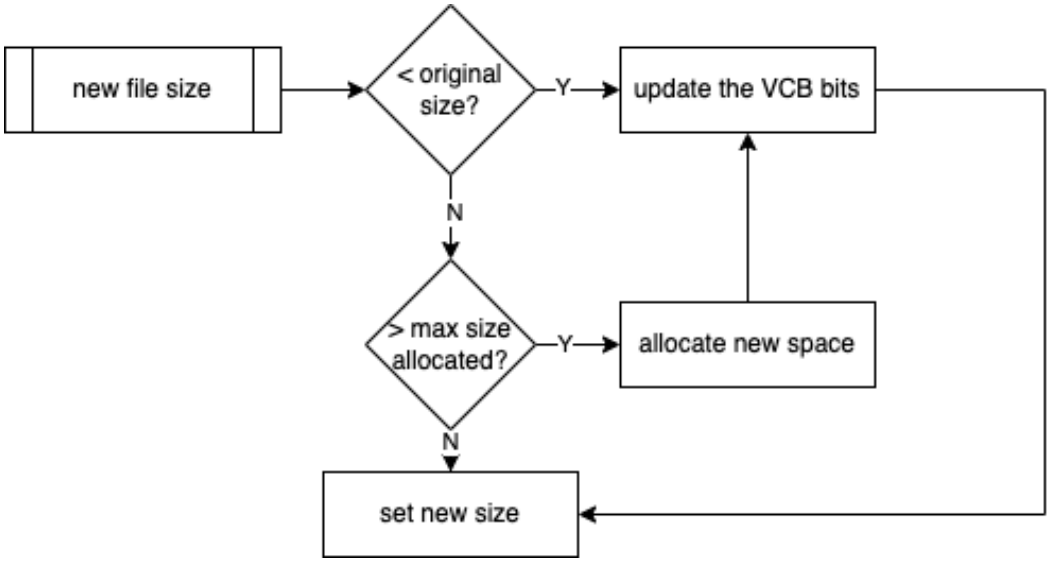
This file write some characters to a file. Notice that the filesystem does not support **append** operation, which means writing is always done by overwriting previous content for a file.

source

When `fs_write()` receives a file pointer, it first checks whether it is valid. If it's not valid, it will print an error and return. Else, there are three scenarios:

1. The new size is smaller than the previous size. We don't need to allocate a new space. However, the valid bits of the file need to be changed (decreased)
2. The new size cannot fit into the previously allocated blocks. This means we have to find a new place to store this file. This process is handled by the `fs_allocate()` function and will be discussed later.
3. The new size is larger than or equal to the previous size, but the previously allocated blocks has enough space to hold the new content. This scenario is quite common in the bonus part, since creating a new file within the directory means the parent directory size need to be modified. In this scenario, there is no reallocation, and the file's VCB bits are remained.

In all of the scenarios, the *size* attribute and the *modifiy time* attribute of file need to be changed, and the file system time need to be increased.

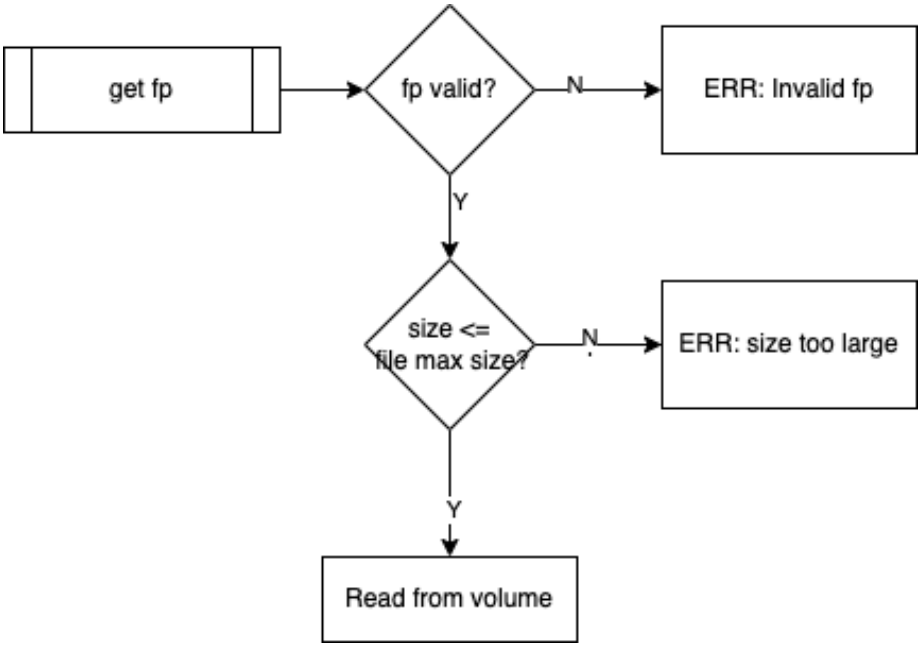


bonus

Aside from updating the *modify time* of the file itself, in the bonus we also have to update the direct parent's *modify time* attribute. Moreover, we also have to update the parent directory's *size* and the file content, because the size of a directory is the number of bits of its contents.

`fs_read()`

The fs_read() is pretty straightforward. It only checks if the fp is valid and whether the read size is smaller than the file size. If so, it reads some bytes from the file's base address.



`fs_gsys(LS_D, LS_S)`

source

These two operations list the files in the root directory based on certain criteria. It is achieved through some sorting algorithm. Since calling host functions (like the ones provided in `thrust` library) is not possible in a *_device_* function, I write my own selection sort algorithm. In both **LS_D** and **LS_S**, two temporary arrays are created, one for storing the FCB index and another for storing the corresponding attribute value (modifiy time/size). These informations are achieved by first traversing all FCBs. Then, I use selection sort to sort both arrays based on the attribute value.

in **LS_D**, since no two files have same modification time, we can directly compare directly. However, if we want to sort based on the file size, we have to compare the creation time of two files to determine the precedence. I do not create the third array for space usage concern. Whenever two files share same size, their creation time is directly read from the volume and compared.

bonus

In the bonus part, instead of listing all the files and directories, we only list the ones present in current working directory. Moreover, a file is differentiated from a folder by checking the *is_dir* bit in the FCB.

`fs_gsys(RM)`

source

In the basic version of the project, removing a file comprises the following steps:

1. Find the FCB index with the requested name
2. Clear the VCB bits
3. Invalidate the FCB
4. Decrement the file number count

Notice that deleting a file acutally preserve the original information in the volume. However, they are not achieveable by the user anymore.

bonus

In the bonus part, aside from the above-mentioned steps, we also have to:

- Prevent the user from using `RM` on a directory
- Pop the file name from the parent directory's content and update the *size* of the parent
- Update the *modify time* of the parent
- Increment the system time

`fs_gsys(RM_RF)` (bonus)

The `RM_RF` can only be used to a directory. It recursively deletes all the files and subdirectories within the directory before ultimitely deleting the folder itself. Therefore, it can be viewed as calling `fs_gsys(RM_RF)` and `fs_gsys(RM)` depending on the type of content.

Lastly, just like `fs_gsts(RM)`, after deleting the directory itself, we also have to update the parent directory's content and size and update the modify time, the number of files and the system time.

`fs_gsys(CD)` (bonus)

This function is simply done by updating the global variable of current working directory and incrementing the level depth count.

`fs_gsys(CD_P)` (bonus)

This function is done by retriving the fp information from the global variable of current woring directory, replacing it with its parent's fp, and decrementing the global level depth count.

`fs_gsys(MKDIR)` (bonus)

The `MKDIR` function is actually calling the `fs_open()` to create a new directory, because in the file system's point of view, a directory is simply a file whose content is the files and subdirectories within this directory. Therefore, in `fs_gsys(MKDIR)`:

1. We have to validate this directory's FCB, set the *is_dir* bit and the *parent_fp* bits
2. Set the *name* , *create_time*, *modify_time* attribute of this directory's FCB
3. Update the parent FCB's contet

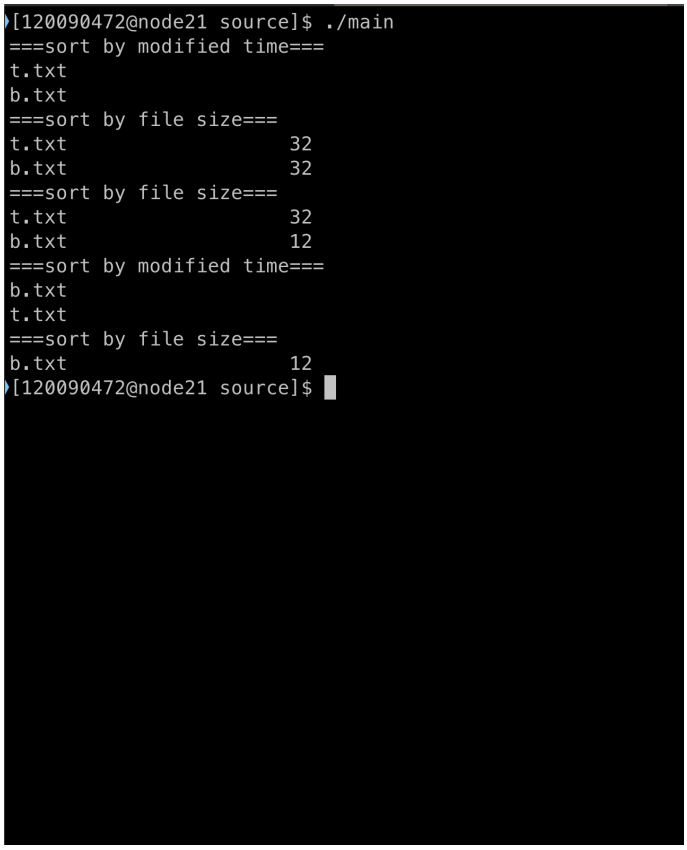
`fs_gsys(PWD)` (bonus)

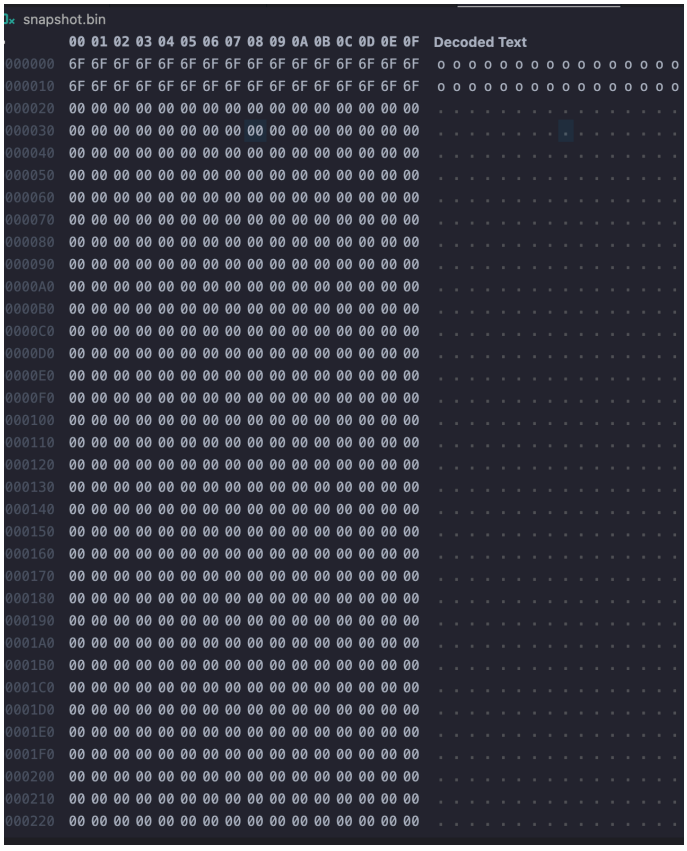
This function is done by looping indefinitely, starting from the current working directory, we can find its name and parent fp, and its parent's parent..... until the root directory. We can construct an array with size global level depth count - 1, because we don't print the root directory unless we are in it.

e.g: if the current working directory is `/app/soft`, we do not print `//app/soft`, but if we are in root directory, we print `/`. Notice we are traversing to the root directory bottom-up, so we have to print the last element in the array first.

Testcase Output

Case 1:


[120090472@node21 source]$ ./main
===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
[120090472@node21 source]$


snapshot.bin
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded Text
000000 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F o o o o o o o o o o o o o o o o
000010 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F o o o o o o o o o o o o o o o o
000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000210 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000220 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Case 2:

```
[120090472@node21 source]$ ./main
===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
```

snapshot.bin

	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded Text
000000	6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F	o o o o o o o o o o o o o o o o
000010	6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F	o o o o o o o o o o o o o o o o
000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0000A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0000B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0000C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0000D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0000E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0000F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000190	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0001A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0001B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0001C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0001D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0001E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
0001F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000200	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000210	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .
000220	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . .

Case 3:

<omitted same part with previous testcases>

```
===sort by file size===
EA                      1024
~ABCDEFGHIJKLM          1024
aa                      1024
bb                      1024
cc                      1024
dd                      1024
ee                      1024
ff                      1024
gg                      1024
hh                      1024
ii                      1024
jj                      1024
kk                      1024
ll                      1024
mm                      1024
nn                      1024
oo                      1024
pp                      1024
qq                      1024
}ABCDEFGHIJKLM          1023
|ABCDEFGHIJKLM          1022
{ABCDEFGHIJKLM          1021
zABCDEFGHIJKLM          1020
yABCDEFGHIJKLM          1019
xABCDEFGHIJKLM          1018
wABCDEFGHIJKLM          1017
vABCDEFGHIJKLM          1016
uABCDEFGHIJKLM          1015
tABCDEFGHIJKLM          1014
sABCDEFGHIJKLM          1013
rABCDEFGHIJKLM          1012
qABCDEFGHIJKLM          1011
pABCDEFGHIJKLM          1010
```

< Omitted size from 1009 to 52>

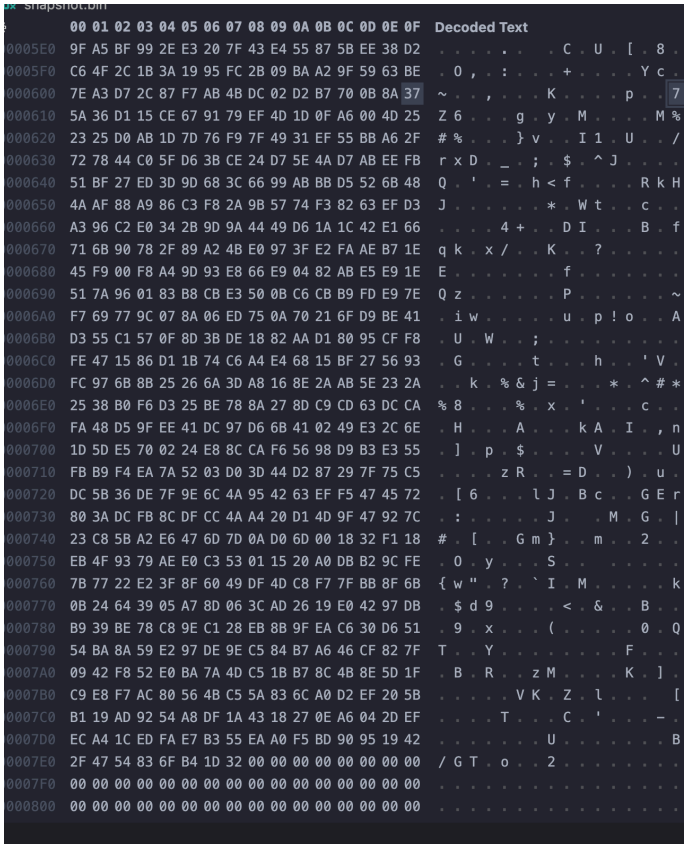
```
MA                      51
LA                      50
KA                      49
JA                      48
IA                      47
HA                      46
GA                      45
FA                      44
DA                      42
CA                      41
BA                      40
AA                      39
@A                      38
?A                      37
>A                      36
=A                      35
<A                      34
*ABCDEFGHIJKLMNOPQR     33
;A                      33
)ABCDEFGHIJKLMNOPQR     32
:A                      32
(ABCDEFGHIJKLMNOPQR     31
9A                      31
'ABCDEFGHIJKLMNOPQR     30
8A                      30
&ABCDEFGHIJKLMNOPQR     29
7A                      29
6A                      28
5A                      27
4A                      26
3A                      25
2A                      24
b.txt                   12
[120090472@node21 source]$
```

snapshot.bin

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
000000    6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F  o o o o o o o o o o o o o o o o
000010    6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F  o o o o o o o o o o o o o o o o
000020    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000030    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000040    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000050    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000060    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000070    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000080    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000090    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0000A0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0000B0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0000C0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0000D0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0000E0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0000F0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000100    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000110    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000120    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000130    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000140    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000150    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000160    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000170    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000180    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000190    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0001A0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0001B0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0001C0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0001D0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0001E0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0001F0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000200    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000210    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
000220    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
```

snapshot.bin

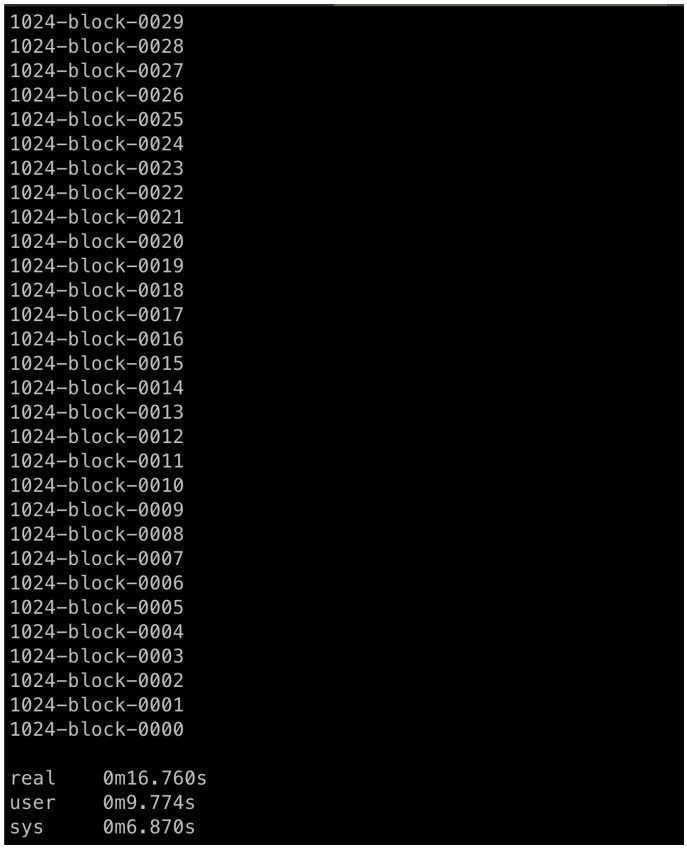
```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
0003D0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
0003E0    00 00 00 00 00 00 00 00 FA E5 17 F8 A2 DC 72 AF  . . . . . . . . . . . . . . r .
0003F0    4B A0 28 C0 C3 18 25 EE E6 9F F4 44 5A 7E 8E E9  K . ( . . . % . . . . D Z ~ . .
0000400   95 C5 E4 24 E3 74 29 DE D9 BF 57 FC 9D CA AC E8  . . . $ . t ) . . . W . . . . .
0000410   6B 54 2A AE EB CE 1D D3 EE 12 97 49 90 A5 B2 A6  k T * . . . . . . . . I . . . .
0000420   6B 97 CA 50 8C 73 AE 66 34 07 63 D1 D1 10 3A BC  k . . P . s . f 4 . c . . . : .
0000430   64 E3 6B 51 B3 88 A4 22 1A BB 6B AA 61 9D 51 CC  d . k Q . . . " . . k . a . Q .
0000440   B5 9C 9C 42 10 4C A8 44 53 0D 95 A4 9C 50 E0 81  . . . B . L . D S . . . . P . .
0000450   B3 CB D2 67 54 F6 89 ED B2 74 98 14 92 6A 60 48  . . . g T . . . . t . . . j ` H
0000460   07 FD 8A 96 4A 33 DB 1D BF F0 41 5D C0 22 DE 75  . . . . J 3 . . . . A ] . " . u
0000470   ED 31 5C C1 28 66 AF 5A DA C7 ED EC B1 4E 35 38  . 1 \ . ( f . Z . . . . . N 5 8
0000480   CB 3F CF 95 F2 2B 32 B2 1C 73 10 DD 95 6E 53 03  . ? . . . + 2 . . s . . . n S .
0000490   1F AF 44 C6 16 F3 21 71 3C 8E DD ED 5D 14 27 29  . . D . . . ! q < . . . ] . ' )
00004A0   53 F6 3F C5 22 71 79 BD E5 09 1B FA F7 ED 7E 17  S . ? . " q y . . . . . . . ~ .
00004B0   1E C2 DE B3 B7 00 A4 F3 8F 83 61 EC 17 88 95 E9  . . . . . . . . . a . . . . . .
00004C0   FE D4 B0 21 47 2A 5F AC 33 7A A7 AA E8 26 C2 07  . . . ! G * _ . 3 z . . . & . .
00004D0   E9 A1 BA 21 21 DF 94 30 63 F5 1D 7A FE 33 64 FD  . . . ! ! . . 0 c . . z . 3 d .
00004E0   87 15 9F CE BE FE FA 72 F8 A3 1D 61 49 DF 68 33  . . . . . . . r . . . a I . h 3
00004F0   81 A3 D3 23 83 E7 53 E6 5E F0 E0 5D 24 C4 5B AB  . . . # . . S . ^ . . ] $ . [ .
0000500   DA 7A FA 19 F8 F5 8B 72 19 A9 D3 63 09 3D 16 0B  . z . . . . . r . . . c . = . .
0000510   60 EA 2E E3 52 81 4A B0 72 2B 0E 16 EF E9 42 4A  ` . . . R . J . r + . . . . B J
0000520   64 BC 64 DD 32 6F 50 4C 98 24 AF A2 61 45 2D 41  d . d . 2 o P L . $ . . a E - A
0000530   AF 5B 25 03 5C EE B3 4F 99 42 65 0A 2C 27 54 10  . [ % . \ . . O . B e . , ' T .
0000540   E3 38 ED 17 28 3E 63 C0 E2 92 63 44 D7 90 06 88  . 8 . . ( > c . . . c D . . . .
0000550   6B 2B 0B C8 9A BE 18 34 80 FC 3E 2C 25 13 3D 09  k + . . . . . 4 . . > , % . = .
0000560   CA AA 20 F2 69 03 B4 4C 95 97 10 ED A8 16 F5 14  . . . . i . . L . . . . . . . .
0000570   42 01 5C DC 3F F3 90 40 F1 CF 6C 17 E2 A9 9F AD  B . \ . ? . . @ . . l . . . . .
0000580   55 C0 A1 BE 43 D5 8A D9 6D 9A 47 95 B1 3D 2A F3  U . . . C . . . m . G . . = * .
0000590   BD 86 50 7C 7B E0 BC 6D 30 2A 04 92 53 A3 C0 28  . . P | { . . m 0 * . . S . . (
00005A0   E3 E1 66 28 B7 F0 81 A4 8C C8 3B 3E 85 65 B1 C2  . . f ( . . . . . ; > . e . .
00005B0   6B 81 BE E6 E1 7C D3 13 26 57 25 79 7B E5 A2 5F  k . . . . | . . & W % y { . . _
00005C0   C7 88 87 7F 7A 88 A4 07 D0 5F C4 D5 C4 76 98 AF  . . . . z . . . . _ . . . v . .
00005D0   F7 58 97 59 54 EA 6C 7A C2 12 73 3E F7 16 9D C0  . X . Y T . l z . . s > . . . .
00005E0   9F A5 BF 99 2E E3 20 7F 43 E4 55 87 5B EE 38 D2  . . . . . . . . C . U . [ . 8 .
00005F0   C6 4F 2C 1B 3A 19 95 FC 2B 09 BA A2 9F 59 63 BE  . O , . : . . . + . . . . Y c .
```

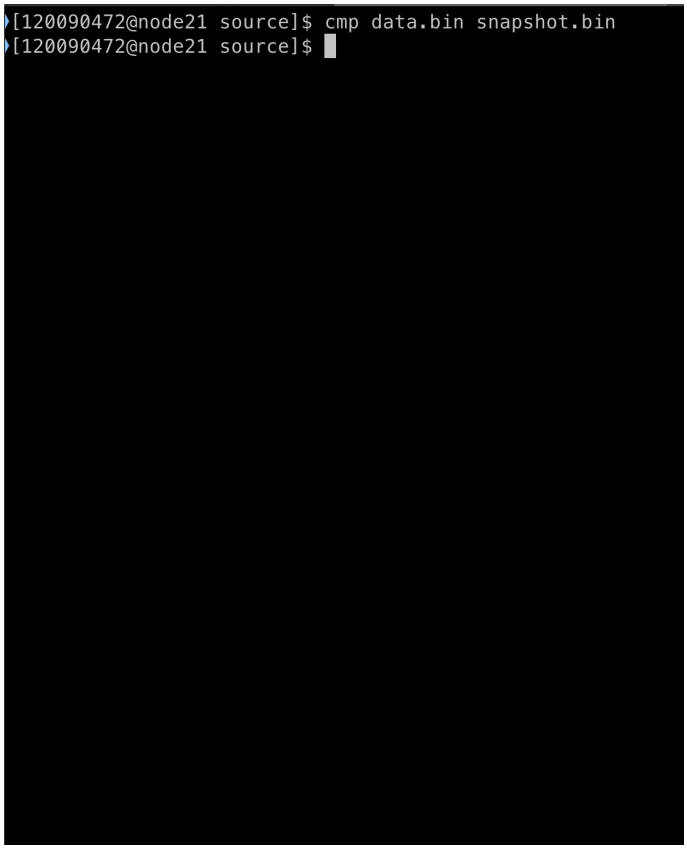


Case 4:

```
[120090472@node21 source]$ time ./main
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
```

<omitted 1024-block-0992 to 1024-block-0030>

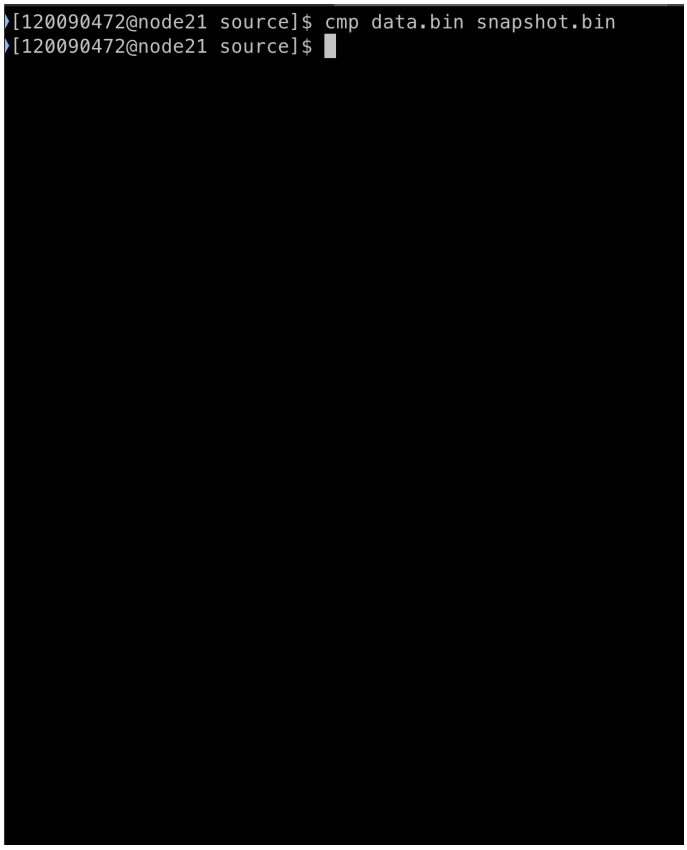

1024-block-0029
1024-block-0028
1024-block-0027
1024-block-0026
1024-block-0025
1024-block-0024
1024-block-0023
1024-block-0022
1024-block-0021
1024-block-0020
1024-block-0019
1024-block-0018
1024-block-0017
1024-block-0016
1024-block-0015
1024-block-0014
1024-block-0013
1024-block-0012
1024-block-0011
1024-block-0010
1024-block-0009
1024-block-0008
1024-block-0007
1024-block-0006
1024-block-0005
1024-block-0004
1024-block-0003
1024-block-0002
1024-block-0001
1024-block-0000
real 0m16.760s
user 0m9.774s
sys 0m6.870s


[120090472@node21 source]$ cmp data.bin snapshot.bin
[120090472@node21 source]$

Bonus

```
[120090472@node21 bonus]$ ./main
===Sort 2 files by modified time===
t.txt                   4
b.txt                   3
===Sort 2 files by size===
t.txt                   32
b.txt                   32
===Sort 3 files by modified time===
app                     5               d
t.txt                   4
b.txt                   3
===Sort 3 files by size===
t.txt                   32
b.txt                   32
app                     0               d
===Sort 0 files by size===
===Sort 3 files by size===
a.txt                   64
b.txt                   32
soft                    0               d
===Sort 3 files by modified time===
soft                    10              d
b.txt                   9
a.txt                   7
/app/soft
===Sort 4 files by size===
B.txt                   1024
C.txt                   1024
D.txt                   1024
A.txt                   64
===Sort 3 files by size===
a.txt                   64
b.txt                   32
soft                    24              d
```

```
/app
===Sort 3 files by size===
t.txt                   32
b.txt                   32
app                     17              d
===Sort 2 files by size===
a.txt                   64
b.txt                   32
===Sort 3 files by size===
t.txt                   32
b.txt                   32
app                     12              d
[120090472@node21 bonus]$
```



Problems encountered

Revealing the status of the file system

In the debugging process, it is often necessary to figure out the current status of the file ststem. Hoever, CUDA-GDB is often not very helpful, as I find some variables cannot be printed out. Therefore, I write a function `fs_diagnose()` that prints all metadata of the valid FCBs.

Allocating new space using `fs_allocate()`

This function simulates the allocation of a file that is too large to fit in the previously allocated block, which could happen if:

1. The file is being written for the first time, and previously has not been allocated with any space
2. The file's new size is greater than the previouly allocated space

To allocate a new space, we adopt **Next Fit Algorithm**. In this algorithm, we use the global variable glastblock, which records the first block after the previous allocation in the VCB. Starting from that position, the program will traverse sequentially, looking for *n* consecutive free bits in the VCB, where *n* is the new block size of the file. When no such block is found, a file compression is needed.

Volume compression using `fs_compress()`

We need copression because our file system generates **external fragmentation** during its normal usage. An external fragmentation happens when we delete a file or overwrite a file with less content. This means that although there are enough blocks available to store files, they are not contiguous. One way to eliminate such fragmentation is done through compressing the volume. In this operation:

1. We first sort all the files in ascending order based on their *start block index* attribute, which is readily acquired from the FCBs
2. Starting from the first file, we loop through the sorted file list, moving the next file's start block next to the previous file's end block. This operation involves the following operations:
    1. Update the *start block index* of files
    2. Update the VCB bits
    3. Moving the actual file content in the storage area, which is achieved via the `move_file` function.
3. Finally, we also have to update the glastblock value.

It should be noticed that fs_compress does not update the *modify time* of the moved files.

Sorting: Time VS Space

In the original desin, I did not allocate any new space when sorting. I traverse the FCB 1024*1024 times. Each time, I find the maxmimum value that is smaller than the previous maximum value and print it out. However, such operation significantly reduces the program's speed. Therefore, in later designs, whenever I need to do sorting, I always use the heap space to store arrays and free them when the sorting is done.

Learning Outcome

In this project, I learned how to design a file system. Moreover, I learned to apply the programming philosophy of abstraction. In a file system, from the user point of view, he doesn't need to know how a file is opened/read/written. Moreover, some functions in my design is basically wrapping other functions. For example, the RM_RF is essentially recursively calling RM to remove files. This project also improves my C++ programming and debugging skills. Although in the end I cannot use the standard library or the thrust library, I still appreciate that the USTFs in the WeChat Group mentioned these librarys. They could be very useful later when I write host functions.