Assignment 4: File System - Report

Name: Qin Peiran Student ID: 119010249

Part 1: Design Approach

1.1 Design of Source

- Overall structure

In my design, I mainly develop three components of the file system. The first one is the file control blocks (FCB), which is used to stored basic attribute of one file (in bonus part, the basic information of directory also stored here) and the pointer points to the physical content address of one file. The second component is the physical memory where stores the contents of files, whose basic storing unit is one block (32B). The third component is the super block which stores the 4KB bit-vector where each bit in it indicates whether one corresponding block in the physical memory is occupied by existed files or not. I will explain how I design these three components in detail.

As for the file control blocks region, there are totally 1024 FCBs where each size is 32B. One FCB stores one file's basic information, including file name (using 20B), create_time (using 2B), modified time (using 2B), address of the head of its block in physical memory space (using 4B) and one valid bit which indicates whether this FCB is empty or not(using 1B). Figure 1 shows the designed structure of one FCB.

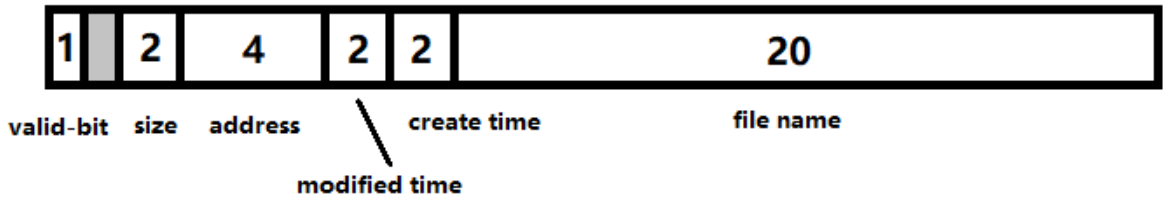

Figure 1: The stucture of one FCB

As for the physical memory region storing files' content, it is devided into 32*1024 blocks and each block occupies 32B. The most important issue of this region is the **maintainance**, including guaranteeing the contiguous allocation, compaction and so on. As for the contiguous, the allocated blocks in this region to one file should be contiguous to avoid the external segmentation. What's more, to make a good use of the memory, the once there is a free block appears, there is a mechanism to do the compaction. Figure 2 shows how this system resolve the problem of compaction. If there free blocks released by one file, the system would first recognize the size of this region by reading the size attribute storing in the FCB. Then, it would move the blocks behind the empty blocks forward. Finally it will update the bitmap storing in the super block region.

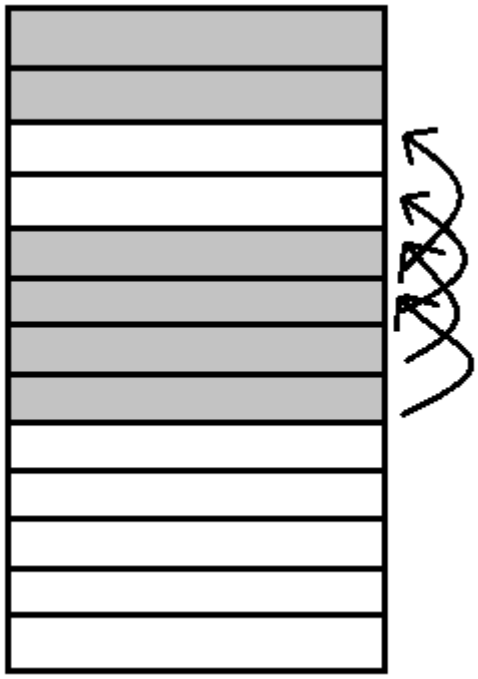

Figure 2: For compaction, move the block behind to the empty block

For the third component, to indicate whether one block in the physical memory has been occupied or not, a bit map is designed. There are totally 4KB bits indicating the blocks one by one. When booting the program, all the bits would be initialized to 0. When the block is assigned to one file, the corresponding bit would be modified to 1 (if it is released by one file, change to 0 again). Figure 3 shows how the bits in bitmap maps to the blocks in content region.

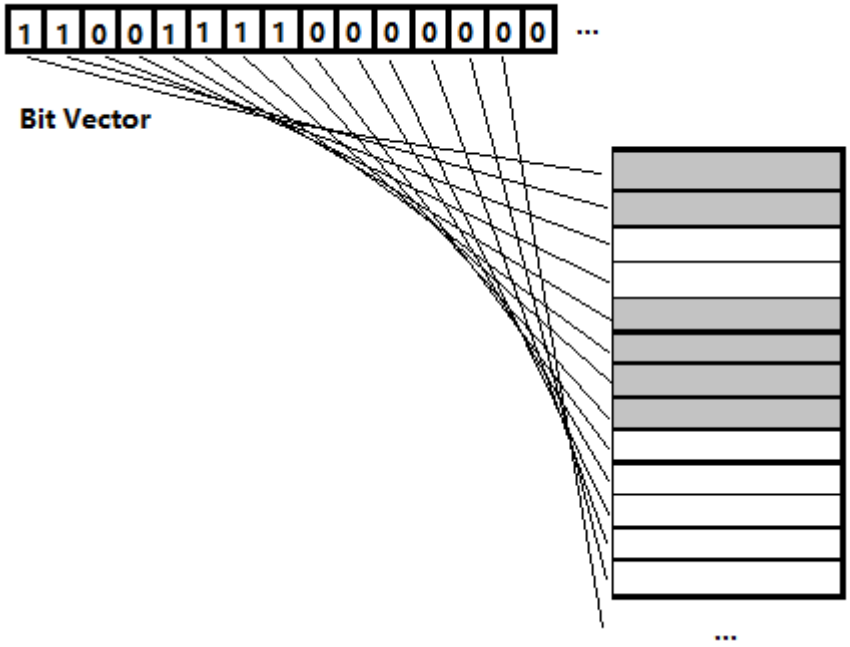

Figure 3: The bitmap strategy indicating the state of block

Based on the aboving design of the three main components of file system, I will explain how these three components coorperate with each other. Figure 4 shows the mechanism of the components' connection. The address stored in FCB indicates the head address of in the physical memory. Then each bit in the bit vector corresponds to one block in contents.

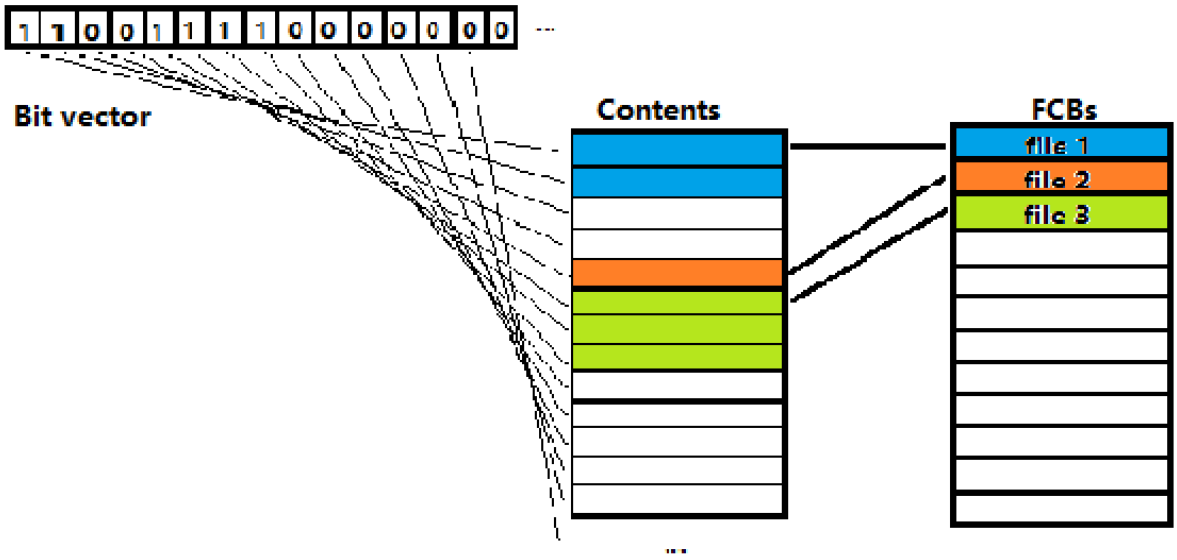

Figure 4: The connection between three main components of the file system

- Implementation of APIs

To make the file system feasible, several APIs are defined, including `fs_open`, `fs_write`, `fs_read`, `fs_gsys(RM)`, `fs_gsys(LS_D)`, `fs_gsys(LS_S)`. I will explain my implementation details of each API.

- **fs_open(Filesystem *fs, char *s, int op):**
  - This function would open a file and return a pointer points to FCB of that file. If the file don't exist, it will create a file, allocate an empty block for it and set its content with an empty byte.
  - First, it will search all the valid FCBs and check whether its name equals to the file name. If it find the FCB with the file name, it will return the address of this FCB back.
  - Then, if it is the file is not found, then it will create a new file by following processes:
    1. Find an empty block in the region of contents, allocate it to the new file and then update the bitmap.
    2. Find an empty FBC, stores the attributes into the FBC, including, created time, modified time(when first create, its modified time would be create time), size (initially 0) and the address of the block we find in first step. Then, set the valid bit of this FBC to 1.
    3. Finally return the address of the newly created FBC.
- **fs_write(Filesystem *fs, uchar *input, u32 size, u32 fp)**
  - This function would use a pointer to locate a file's FCB, then it will find check whether this FCB is valid or not.
  - After that, it would retrieve the size of the file and check whether the current assigned block(s) are not small or large for writing the input buffer to the content. If the input data acquires less or more blocks than the already assigned number of blocks, the number of blocks should be re-allocated.
    - To re-allocate the blocks, in my program, I first delete the original file using `fs_gsys(RM)`, which will be explained later (the compaction of data is also included in this API). Then using `fs_open()` to create a new file. Since the compaction has been done during the `fs_gsys(RM)`, `fs_open()` must allocate the new file with the last unused block in the region of contents. Then, the remaining blocks the file needed is assigned right after the block `fs_open()` allocating to the file.

        - After the re-allocate, the number of blocks the file owned would match the number of blocks it needed to contain the data in input buffer. Finally, we write the bytes in the input buffer one by one to the blocks of content region.
- **fs_read(Filesystem *fs, uchar *output, u32 size, u32 fp):**
    - In this API, we directly use fp to locate the FCB, and retrieve the address to the content region from FCB then read the data one by one to the buffer.
    - Since the size it wanna read may be larger than the actual size of the file, this situation would be handelled to ensure the robust.
- **fs_gsys(RM):**
    - The implementation of this API includes the following procedures:
        1. First it would find the corresponding FCB according to the name of the file.
        2. After the FCB is founded, it will first retrieve the address of head of block in the region of contents and the size of the file. We stores these before delete the file because it is needed when performing compaction.
        3. Then, it clears all the things in FCB meanwhile sets the valid bit in this FCB back to 0. Finally, it would use the size and address of block to do the compaction. The brief mechanism is shown in Figure 2, which will first recognize the empty blocks and move the blocks after them forward as well as the update the bitmap. What's more, it also needs to update the FCBs because some blocks has been moved forward during the compaction. To ensure the consistency between FCBs and the region of content, the address attribute in FCB should be modified according to the how much the corresponding blocks are moved during the compaction.
- **fs_gsys(LS_D / LS_S):**
    - To reduce the memory cost of the file system, during this implementation, I doesn't malloc new memory space to do the sorting. Instead, I first find the number of files I need to sort for, then use two for loop to find the file with largest size (or newest time) one by one.

1.2 Design of Bonus

- In bonus part, I mainly handels all the operaions required by building a **Directory control blocks**. The Directory control blocks act mainly two functions: 1. maintain the all the operations relative to the directories. 2. coorperate with FBCs to form a secondary structure, which will enhance the speed of operating on a file.
- Before explain the design of the Directory control blocks, I will first characterize the structure of directories. First, the information of directories will also be stored in the FCB as files do. The attribute of size is set to the sum of character bytes of all files name. The valid bit will be set to 3, in order to distinguish it with files. Then, tree structure is adopted, and Figure 5 inllustates it:

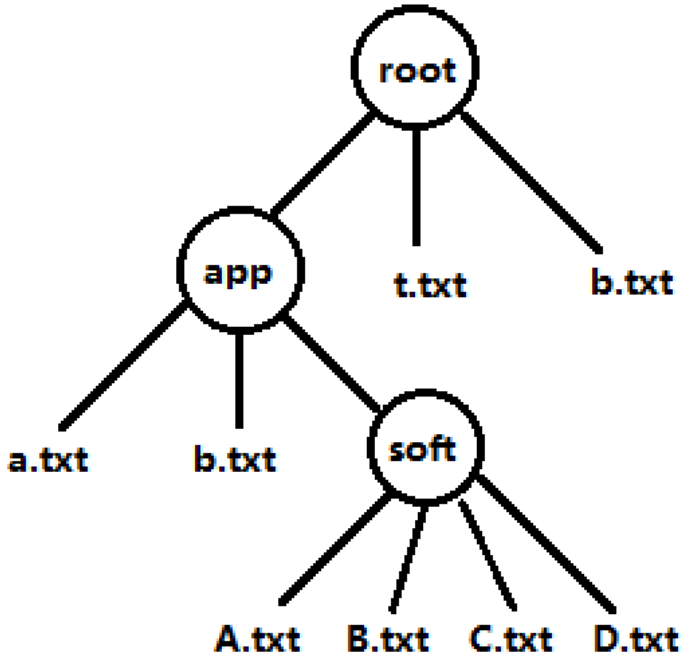

Figure 5, Directories' tree structure

- Moreover, I would explain my design idea of directory control blocks. One directory control block contains the information about one directory's father directory, its address of FCB and all the address of FCB of the files under it. For instance, the arangement fo the directories control blocks would be like:

	Pointer to its FCB	Pointer to its father dir	Pointer to file1 FCB	Pointer to file2 FCB	...	valid bit
						0
soft	FCB of soft	Pointer to app	FCB of A.txt	FCB of B.txt	...	1
app	FCB of app	Pointer to root	FCB of a.txt	FCB of b.txt	FCB of soft	1
root			FCB of t.txt	FCB of b.txt	FCB of app	1

Figure 6: an example of the structure of directory control blocks

- Except for support functions related to direcotories, with these directory control blocks, my file system also forms a secondary structure which can enhance the operation on files. For example, To locate a certain file, the system would no longer need to search all the files in the FCBs. Instead of that, it could recognize what directory the user is currently in, then use the pointers to the FCB stored in one directory control blocks to search. This appraoch can narrow the selections of FCBs and perform a faster performance.

- Implementation of APIs

In Bonus part, I implemented the following APIs : `fs_gsys(MKDIR)`, `fs_gsys(CD)`, `fs_gsys(CD_P)`, `fs_gsys(RM_RF)`, `fs_gsys(PWD)`, `fs_gsys(LS_D)`, `fs_gsys(LS_S)`. To better implement these functions, I add some new attributes to the class of File system.

1. fs->directory_control_memory: This is a memory space I allocate for the implementation of the Directory control blocks mentioned above.
2. fs->current_dir: Indicate what directory the user is currently in.

With added this preparations, I would explain how the program perform the following APIs in detail:

- **fs_gsys(MKDIR)**
  - To create a new directory, I modify the `fs_open( )` implemented in the Source part a bit to make it also support creation of directory. Much same as the creation of files, creation of directory also first find an empty FCB and stores the information into the FCB. The differences is creation of directory don't need to allocate a block for it in the physical memory of contents. Instead, it would find an empty directory control block, store the address of the the current directory and the address of its FCB, then set the valid bit to 1.
- **fs_gsys(CD)**
  - Since I add attribute `fs->current_dir` to the file system. The implementation of `CD` is first searching for the target directory in the directory control blocks, then assign the new address of directory control block to `fs->current_dir`.
- **fs_gsys(CD_P)**
  - Because the directory control blocks also stores the information of one block's parent block's address, CD_P retrieve the parent address of directory control blocks and change the `fs->current_dir`.
- **fs_gsys(RM_RF)**
  - The implementation of this API is quite hard because it not only delete the files and directories, but also need to update the information stores in the directory control blocks. What's more, RM_RF also needs to consider deletion of the sub-directories under the target directories. The implementation is devided into two parts: 1. if there is sub-directories, delete the sub-directories under the target directory, 2. if all the sub-diretories are deleted, then deleted all the files under the directory. Figure 7 illustates this idea with deletion of app\0:

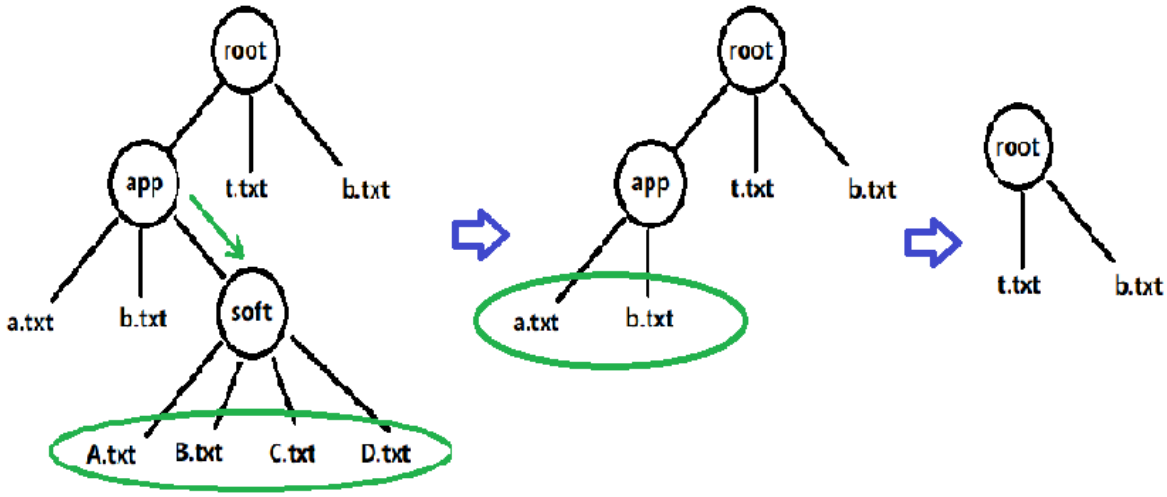

Figure 7: procedures of deleting one directory

When deleting app\0, it will recognize that there is a sub-directory. So it enters the sub-directory and delete all the files(via a for loop with gsys(RM) implemented) in it as well as delete the sub-directory itself. Then, app\0 becomes the directory only contains files. Continue to use for loop with gsys(RM) would delelte all the files under app\0. Finally, delete app\0 itself. Remark: every deletion of one file should also modifies the corresponding

FCB(reduce it's size by the removed file name) and corresponding directory control blocks(remove the corresponding fcb address of removed file).

- **gsys(PWD)**
  - Via `fs->current_dir`, we can find the directory control block of current directory. Since the directory control block also stores a pointer points to its parent, we can use it to futher find its parent until the parent is the root directory.
- **gsys(LS_D / LS_S)**:
  - The principle is similar with the ones implemented in the Source part. The difference is that, these would find the current directory block and narrow the comparation among the address of files in it.

Part 2: The problems I met in this assignment

- Maintain the consistency among different components is the biggest problem I met in this assignment. For example, during the compaction of blocks, not only I should modify the memory region of contents, but also I should modify the FCBs and bit-map (In bonus, directory control blocks should be also modified.)
  - The solution I get is focusing on the key attibutes of a file and stores them during the whole procedure of operation. The key attributes includes the removed file's FCB address, file size, and the address of the head of its corresponding content block. When modify different components, keep in mind with these attributes would make the problem less complex.
- The second problem I met is the handle of fs_write( ) when the input size requires less or more block s the file already assigned.
  - The solution I get is first compare the number of blocks it requires and actually own. If the sizes are not matched, remove the files, doing the compaction and finally re-allocate new fcb and number of blocks.
- The third problem I met is the design issue of the bonus part. I originally want consider about modify the structure of the FCBs to make it support function of directories. However, the problem is that I have already use 31 bytes among the total 32 bytes in the FCB. Adding new attributes to make it avaliable for directory operations means I need to use less bytes to store other attribute, which may cause fatal problem when the size is large. In addtion, allocate a new space for directory control blocks would make the structure clear for me to design as well as speedup the file operations since it forms secondary structures. Facing this trade-off, I finally try the current design issue.

Part 3: Steps to execute my program

Remark, there is a shell script name `run.sh` for compiling.

For Source part:

```
$ cd Source
$ sh run.sh
$ ./main.out
```

For Bonus part:

```
$ cd Bonus
$ sh run.sh
$ ./main.out
```

Part 4: Screen shot of my program

- Part of screen shot running Source code, with provided user program - Test 3

The beginning of output

```
===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
```

The end of output

```
OA 53
NA 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
```

- Screen shot of bonus part:

```
bash-4.2$ ./main.out
===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
```

Part 5: What I learned from this assignment

1. The basic concept of file systems. Through doing this project, I have a more comprehensive understanding of the file system. I acknowledge how the file systems works by the coorperation of FCBs, physical memory of contents and bitmap. What's more, I also learned the design structure of the FCB which stores significant attributes within 32 bytes.
2. The programming skill related to CUDA and building of file system. During the programming, I met several bugs. For CUDA, I thinks its difficult part for programming is the operation related to memory, since CUDA get a more complex memory structure. Therefore, I should carefully consider the memory issue(for example, I should reduce or avoid the usage of recursion). And for the programming skill related to the file system, I make a most of my time doing this assignment in debuging and learned many details when building a file system (including how to better perform the bit operation in bitmap, how to handle the compaction and so on). This experience would be useful If I gonna implement other file systems.
3. Through this assignment, I also learned how to design some details of file systems on my own especially for the bonus part. Since there is no standard criterion for the implementation of file system, many design issues should be considered by our own. For

example, to handle the directories issues, I have to design the directory control blocks and manage them carefully. This experience provides me a better understanding of file systems.