Report for csc3150 assignment 4

How did I design my program?

(i) *How did I implement file volume structure, bit map and file control block?*

I divide the 1060KB volume into three parts, namely, bit map, file control block and physical storage. Bit map is the first 4KB. FCB (short for file control block) comes after bit map, it is divided into 1K blocks, with size for each block is 32Bytes, so fcb's size is 32KB. Physical storage is divided into 32K blocks, each block is 32 bytes, so the physical storage is the last 1024KB in volume. About bit map, since this assignment requires contiguous allocation, which means that the files stored in the physical storage will always be compact, therefore I only need to know where is the last file stored then plus 1, i.e., the first available (unused) physical storage block number. I use *most_adv_0_block* to record this variable. Therefore, I do not need to use bit map. So, the first 4K in my volume is not used. About file control block, first, since each file will occupy 1 block, which is 32 bytes, for storing meta information. I use first 20 bytes to store file name, and the 21th byte to store the read/write mode, range is just G_WRITE and G_READ, and use the 22-23th bytes to store file's location, in which the 22th byte stores the first 8 bit of the location, and 23th byte stores the last 8 bit of the location (since at most we have 1024 files, so we need 10 bits to store the location, one byte is less than 10 bits, 16 is larger than 10 bits, therefore we need 2 bytes to store location), and use 24-25th bytes to store file size (since the biggest file size is 1024 bytes), and use 26-27th bytes to store file's modify time, for fs_gsys(LS_D)'s use, and use 28-29th bytes to store file's create time, for fs_gsys(LS_S) use (if the file size is same, then first create first print), and the 30th byte to store if it is USED in the sort (in fs_gsys(LS_D) and fs_gsys(LS_S)). The remaining 2 bytes are useless.

About FCB, Second, I also do compaction on FCB, like I do in physical storage. The reason is for convenience. Imagine the case that you have 1K files stored in physical storage, now you want to delete a.txt, you should first go to FCB to find where this file locates in the physical storage, i.e., the first block number where this file locates, at the same time you can get its size from FCB. Now it's time to do compaction in physical storage, but you don't know how many files and which files are located behind a.txt in physical storage, so you can only go to FCB and find out the most neighbouring file where its location is larger than a.txt' location, and move this file up by a.txt's occupied block number, update the location for this file in FCB, and continue to find the second file located behind a.txt, and do the same thing as for the first behind file, so and so forth, until you reach the last file in physical storage. It is a huge workload! So I come up with the idea that I should also do compaction for FCB, namely, when I do compaction for physical storage I also do compaction for FCB, in this way I can guarantee that the file locating order in FCB is the same as that in physical storage. Therefore, when I encounter the above case, I just move the files behind a.txt up by a.txt's occupied block number and go to FCBs that are behind a.txt's FCB and update location in them, i.e., I only need to search for a.txt's FCB block, no need to search for other FCBs. Therefore, I also need a variable to store the first available FCB block number, i.e., the existing file number, which is *most_adv_0_fcb*.

(ii) How did I implement contiguous allocation?

I will divide this part into three components, namely, when I create a file, delete a file and write a file (since write a file means first delete it and then create it but with the same file name and create time).

Create a file: since I have done the compaction of both physical storage and FCB, when the user wants to create a file, I simply allocate the first unused FCB for it (use *most_adv_0_fcb*), write in the file's name, size (is 0), create time and UNUSED byte (for sort). Then in fs_write() (since we could not read an empty file, so it must followed by fs_write()), I know how many size will be written, then convert it to occupied block number, then write the file content in physical storage, starting from *most_adv_0_block,* and then update the *most_adv_0_block* by incrementing occupied block number, and update this file's FCB, write in its size and location (which is the starting physical block number this file locates). In this process, I use the fact that FCB and physical storage are both compacted and have the same file order, and I also keep this structure during this process since a newly created file will always has its content and FCB stored in the last.

Delete a file: when user want to delete a file, which is fs_gsys(fs, RM, filename), so we only get the file name that user wants to delete, we need to first go to FCB to search for the matched FCB block. Once find, from the FCB, we can get its size and then convert the size into occupied physical block number. Since both physical storage and FCB are compacted and have the same file order, to know which files are located behind this deleted file in physical storage, we simply go to FCBs that are located behind this deleted files' FCB block and update the location in them by decrementing the deleted file's occupied block number. Then in physical storage we simply move up the files that are behind deleted file by occupied block number of block and in FCB we simply move up the FCBs that are behind deleted file's FCB block by one block, in this way we successfully compact both FCB and physical storage and keep the same file order between them when deleting a file.

<u>*Write a file:*</u> since when the user wants to write a file, we need to first delete its original content and rewrite it, this is the same as first delete a file and then create a file with the same file name and create time. The only difference is that we need to update the modified time. Since it is the combination of the above two parts, I will not restate it here.

<u>*(iii) How do I handle file's created time, modified time and do sort (in fs_gsys(fs, LS_D) and fs_gsys(fs, LS_S))?*</u>

I create two int global variables, namely, *create_time* and *modify_time* to record and update files' created time and modified time. They work like this: they are initialized as 0. Whenever the user creates a file, the *create_time* will increment by 1 and be assigned to be this file's created time. Therefore, first created file will have smaller *create_time* than later created file. Whenever the user writes/rewrites a file, the *modifiy_time* will increment by 1 and be assigned to this files's *modifiy_time*. Therefore, more recently modified file will have larger *modify_time* value.

When I do sort in fs_gsys(fs, LS_D), I first find the file with largest *modify_time* value, then set its UNUSED byte (which is the 29^{th} byte in a file's FCB) to USED, print out its file name, then among the remaining files with UNUSED byte, I continue to find the file with the largest *modify_time* value and print out its name, so and so forth, repeating the procedure. This loop ends when we have sort all files (i.e. the loop loops for number of existing files times), so this is an O(N^2) loop. It is the same for fs_gsys(fs, LS_S), the only difference is that when two files have the same size, then first create first print. So we need to compare the *create_time* of two same size files and choose the file with smaller *create_time* as the larger one and first print it. It is also an O(N^2) loop.

What problems I encountered in this assignment and how did I solve it?

First, I encountered the problem that when the user wants to delete a file and I want to keep the compact structure of physical storage, how can I know how many files and which files are located behind deleted file in physical storage? As I have said before, it is too messy to use random FCB, so I come up with the solution that I should keep both FCB and physical storage compact. A good consequence is that they can have the same file order (i.e., file information and file content are stored in same order), which is very convenient.

Second, when I debug my program, test case 1 and test case 2 are good to go, however, in test case 3, my program will always have some files losing, and it seems like printing error since there are some messy code interrupting the printing. I trace my code thoroughly and still find no mistake. So I ask others, and they told me not to print out byte by byte, but to print in string form. I tried, and it really works! I find it very strange, and ask the reason, and the answer is that it is a bug in CUDA programming language. After all, it's not my fault. However, I find it very annoying, since it takes me one day to solve this problem.

Environment of my program:

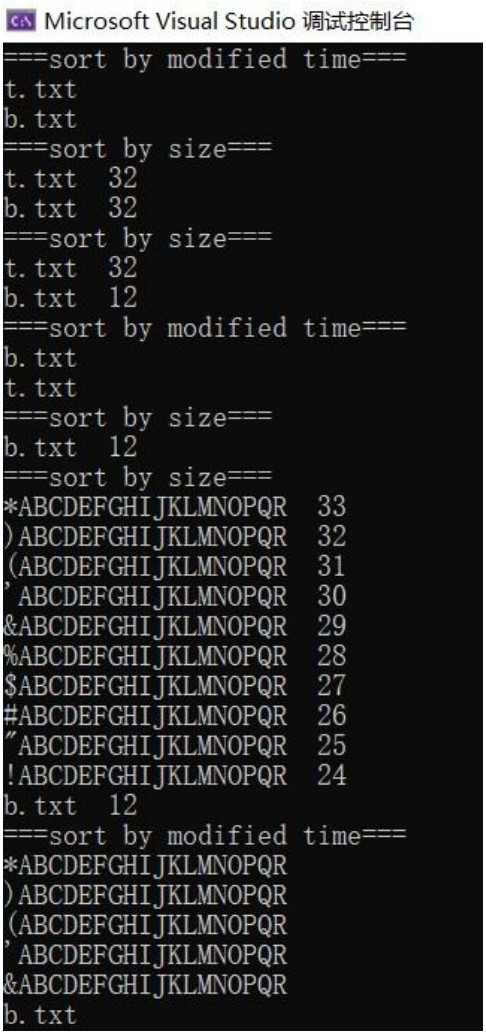
Windows 10, VS2017, CUDA 10.2

Step of running my program:

You simply load my CSC3050_A4 folder into VS2017 (by clicking CSC3050_A4.sln), and press ctrl + F5, then you can see the result.

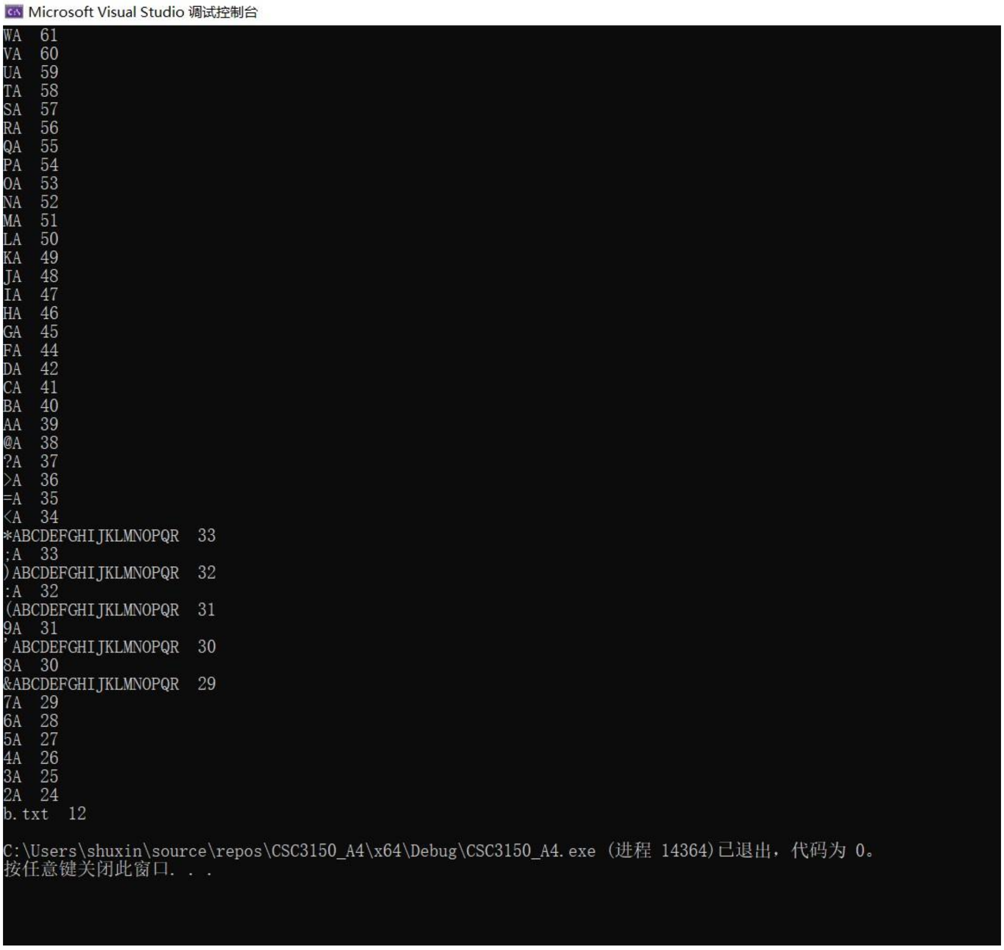
Some screenshots of my program:

Test case 1 & 2:



Test case 3:

```
===sort by 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
```



What did I learn from this assignment?

I learned how a simple file system works.

I learned how to implement contiguous allocation.

I learned how to do compaction.

I learned that sometimes my program has fault does not mean that I am wrong, but something wrong with the programming language. I need to check that what I think it is is not what it actually is.