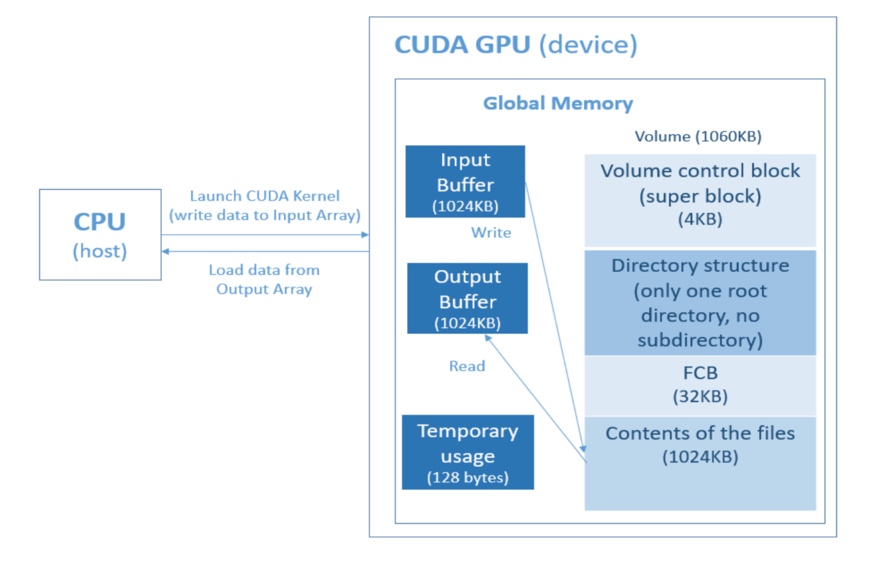
***Report for CSC3150 Assignment 4***

Yang Liang 120090874

1. **Program Design Methodology**

In this assignment, we are required to simulate the mechanism of a file system. The size of the Volume Control Block is 4KB, which is equal to 2^15 bits, hence can be a bitmap that represents the position of each storage block (there is also totally 1024KB/32B = 2^15 storage blocks). The FCB stores the metadata of the file except its detail content, we have totally 32KB of FCB space for at most 1024 files. Therefore, we have 32 Bytes for the FCB of each file. The allocation of this 32 Bytes is as below.

The name for the file occupies 20 bytes, as indicated in the assignment, the “valid” bit of the file occupies 1 byte, the “empty” bit of the file occupies 1 byte (we can use the empty bit (1 empty, 0 non-empty) to efficiently check whether this FCB block has been used), the “start bit” occupies 2 bytes, which stores the index of the first storage block that stores the content of this file (we use contiguous allocation), the “create time” bit occupies 2 bytes, which stores the relative time that the file was created, the “modified time” bit occupies 2 bytes, which stores the relative time that the file was modified, and the “file size” bit occupies 4 bytes (actually 3 bytes is enough), since the maximum size of the file is 1024KB, which is equivalent to 2^20 bits, 3 bytes (equivalent to 2^24 bits) is actually sufficient to store this number information. Therefore, 32 = 20 + 1 + 1 + 2 + 2 + 2 + 4. Moreover, the “volume” is represented as an array with 1085440 elements, each element is a char of 1 byte, hence, we can think of the “volume” as byte-addressable.



**1.2 Implementation Details**

Totally we are required to implement 5 functions.

In *fs\_open()*, we first need to determine the mode by the “op” parameter passes in. We search the given file name in the FCB blocks to retrieve the index of its FCB, by the *search\_file\_name()* function I defined. If there is no match and it is in the write mode, we need to manually allocate an empty FCB block (by the “valid bit”), and search the 1024KB file content space for an empty block (by the bitmap). We need to update its information in the FCB, including plugging its name into the name bit, setting the empty bit to 0, rewriting the start bit as the index of the first empty storage block we find in the file content space, and updating its create time and modified time, both as the current “gtime” value, which is a global variable defined in the template. Moreover, even though the size of the new file is set to 0 byte, as required, we still allocate one whole storage block for it for latter usage, and the bitmap for this block is also set to 1. This function returns a file pointer, which is a 32-bit binary number, with 2-bit as the valid bit (indicating the “read” mode or “write” mode), and 10 bits storing the index of the FCB block that stores the information of this file in the FCB space.

In *fs\_read()*, we read designated bytes of contents from the head of the input buffer (must start at the beginning), to the output buffer. We first decode the FCB index information from the file pointer that passes in as a parameter, to get the FCB block of the read file, and then obtain the index of the first storage block that stores the file information from the FCB block. The absolute byte address of the content can be calculated from the start index of the storage block, by

fs->FILE\_BASE\_ADDRESS + block\_index \* fs->STORAGE\_BLOCK\_SIZE,

we then read the information from the input file to the output block array as we usually do, to retrieve information from an array, since the “volume” here is in essence an array, with the total size as its total number of elements.

for (int i = 0; i < size; i++){

output[i] = 0;

output[i] = (fs->volume[actual\_address + i]);

}.

In *fs\_write()*, we need to write contents from a specific file to the target file. First, if there are already originally some contents in the target file (this information can again be retrieved from the size bit of the FCB block of this file), we need to remove it. Apart from removing it, in my implementation, I modify the bitmap such that all the storage block that originally stores the content of this file except its start storage block index is set to 0. We also change the modify time information of this target file, then increment *gtime* by 1.

If the size to be written is smaller than 32 bytes, writing process can be implemented directly, since every file will surely contain 1 storage block that belongs to it (as my implementation above). If the size is greater than 32, we first need to go through the bitmap, from the position of the start storage block that belongs to it, and check how many empty storage blocks are available after its “start” block. If the number of available blocks is greater than the blocks demanded by the size, writing process can also be directly occurred.

However, when there are no enough empty blocks, we have to do compaction. The big picture of the compaction I implemented is basically searching from the beginning of every block, and compact all files at one time. Whenever there is an empty block (we denote the index of this empty block to be *“empty\_index”*), we first then continue searching the blocks after it, if there are nonempty blocks after it, meaning that there are files after it, also meaning there are “holes” between 2 files, we need to compact, by moving the file that after it to the position of that empty block. We need to update the FCB information of this block that we move ahead for compaction, and the bitmap information. Every time we compact 1 file, after compacting this file, we move forward from the position of *“empty\_index*”, finding another empty space, which is likely to be a *“hole”* among files. But basically, if there are no nonempty blocks after this “*empty\_index*”, the compaction process can be terminated, since there are no files after it, this is the end of all files. After compacting all the other files except the target file that we want to write information, the bitmap will be turned into a pattern that having 1’s at the beginning and 0’s at the end. Hence, we can append the target file at the tail of the other files, then there must be enough space to hold its content (whatever size). We do this by searching the bitmap after compaction to look for the first 0 bit, which is the first empty space of the file content system, then this will become the start index of the storage blocks that stores the contents of the target file to be written, we write contents to it afterwards.

*fs\_gsys(RM)* is quite trivial. We first invoke the *search\_file\_name()* function to find the FCB index of the file to be removed. We can then find the absolute byte address of this file and remove the content of the file, by initialize them to 0, byte by byte (index by index). Then we delete its FCB block, by again initializing them to 0, and set the valid bit to 1 (since as earlier definition, 1 means empty). Next, we update the bitmap to set all the blocks that are originally assigned to this file to 0.

*fs\_gsys(LS\_D/LS\_S)* requires us to list the information about the files. LS\_D lists all file names and order by its modified time of files. LS\_S list the files names and size order by size. If there are multiple files with the same size, the one with smaller create time is put first. These sorting criteria information are all in the FCB, which can be easily retrieved. To sort them, I implement the bubble sort algorithm, first, I establish an array that consists of the FCB indexes as contents, since not all FCB blocks contains file information (some FCB blocks are empty). We then do bubble sort based on the contents that can be retrieved by the FCB indices that are stored in the array.

**1.3 Bonus**

We are required to implement the file structure in tree directories. Some changes can be made to the FCB data structure, we need 1 bit to indicate whether it is a directory or a file (this can be put to replace the position of the “Valid bit”). Since directory is also a file, we may use 3 “pointers” in FCB, one stores the FCB index of its next file, one stores the FCB of its parent directory, and the third is the FCB of its child (only for directory), all of the pointers are size 1 byte.

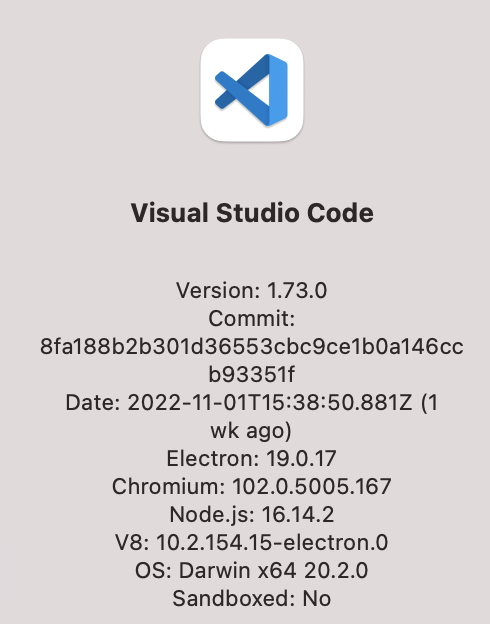
I will implement the mkdir() and fs\_gsys() function,

The other detailed implementation, please have a look at my codes, and give some points if possible.

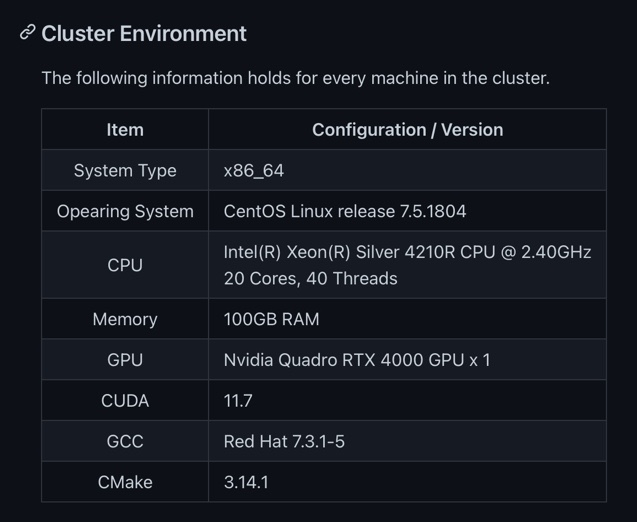
1. **Program Execution Environment**
   1. **OS**



* 1. **VS version**



* 1. **CUDA Version and GPU information (Cluster Environment)**

****

1. **Program Execution Commands**

Please use the commands below in the terminal to execute the program in the cluster, for both the main task and the bonus.



1. **What is Learned from this Assignment**

I think this is the assignment that has longest lines of codes in this course. And I know more about the file system, especially when struggling with the file compaction process. Of course, writing so many lines of codes indeed improves my C programming skills, and generally, the process of finishing this assignment is full of complication, I met many weird problems, such as the weird outputs if we do not initialize first before writing anything to the “volume” array…

1. **Problems Met in this Assignment**
2. Understanding the concepts

Read the textbooks, discuss with classmates, listen to the lectures and tutorials

1. Debugging issues
2. Manipulating memory address using operators such as &, | (should be very careful with this, since I accidentally write the wrong direction for the shifting operators for several times)
3. Printing of names seems not convenient, and some pitfalls occur if we print them char by char. After consulting fellow classmates and Piazza, perhaps this is because of the CUDA issues, we had better store the char into an array and print them by

printf(“%s\n”, name\_array).

1. The implementation of the compaction is the most challenging part in my opinion. We have to clarify the logic and make myself clear. We need to be careful of the sequences of modifying the bitmap when compacting the files (first set 1 in the bitmap to 0, then set 0 to 1), otherwise, problems may occur.
2. We must initialize the contents in the “volume” array before writing anything to it, in each iteration, like below,

for (int i = 0; i < size; i++){

output[i] = 0;

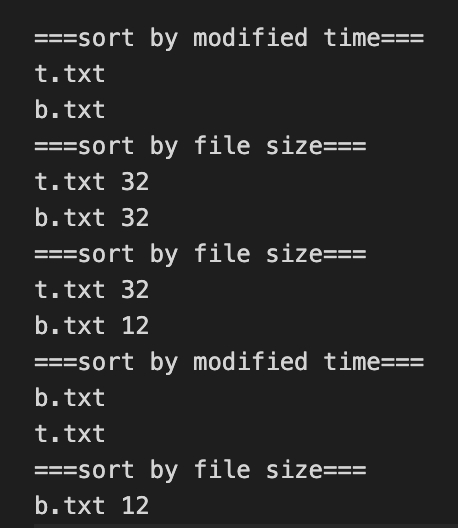
output[i] = (fs->volume[actual\_address + i]);

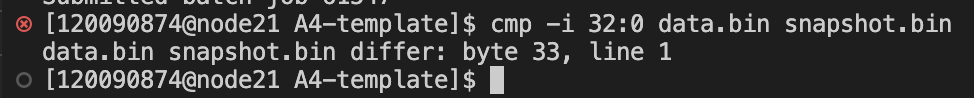
}.

otherwise, strange things occur. This makes me really confusing and wastes me a lot of time, I don’t know why…

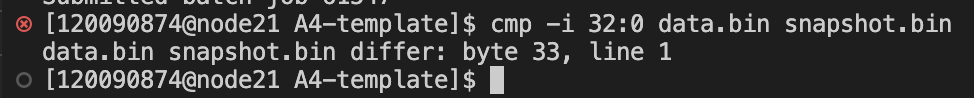
1. **Screenshots of the Program Outputs**

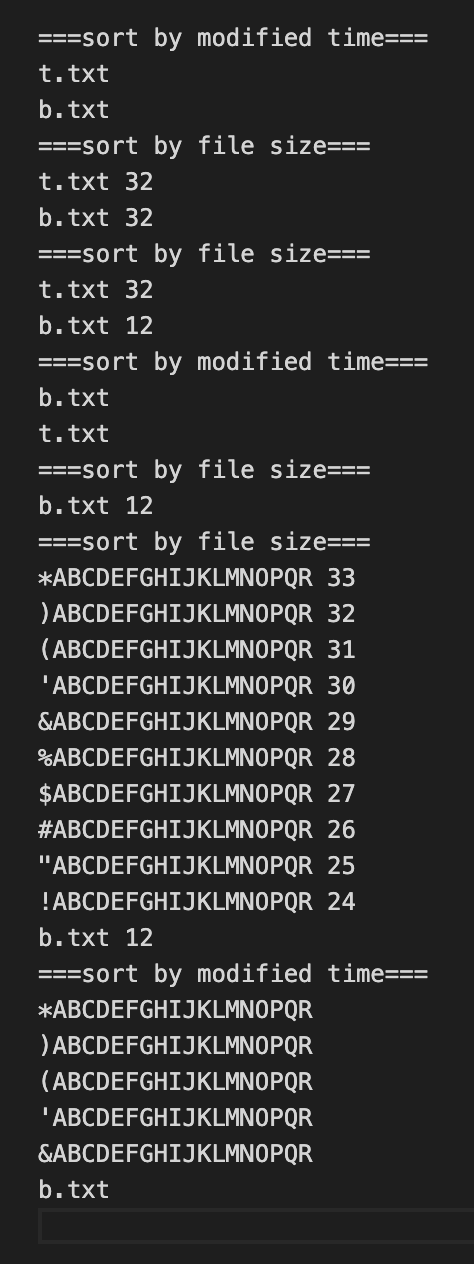
**Task 1**



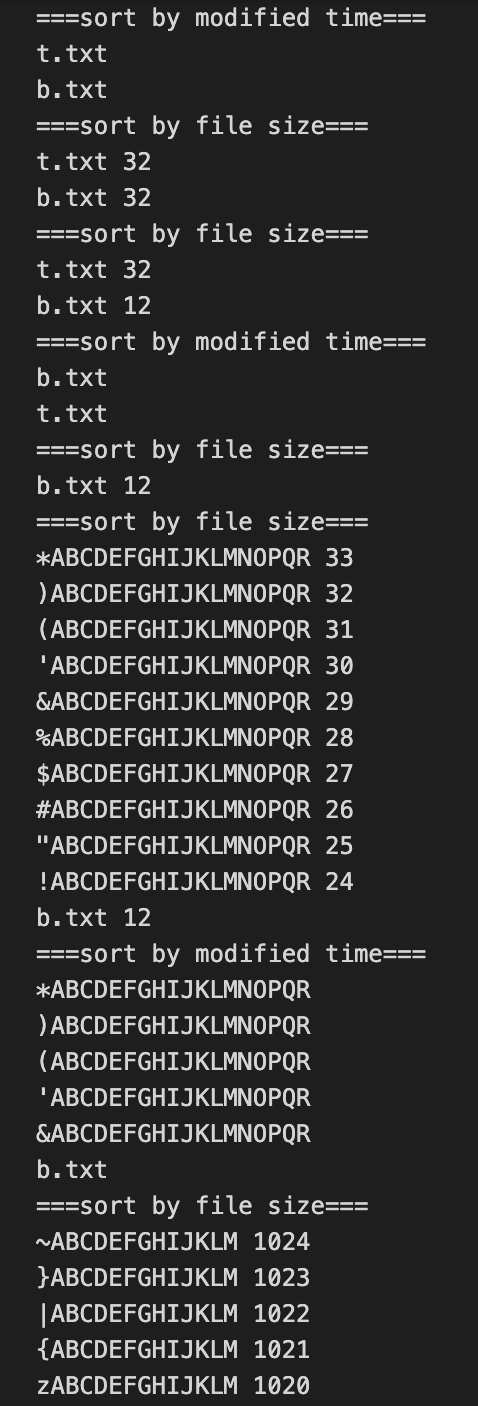


**Task 2**

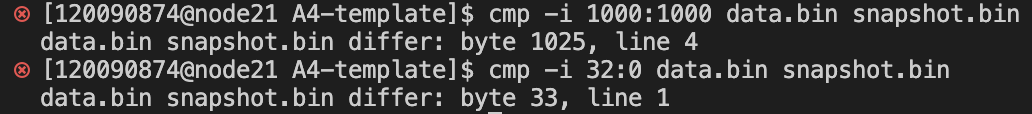




**Task 3**



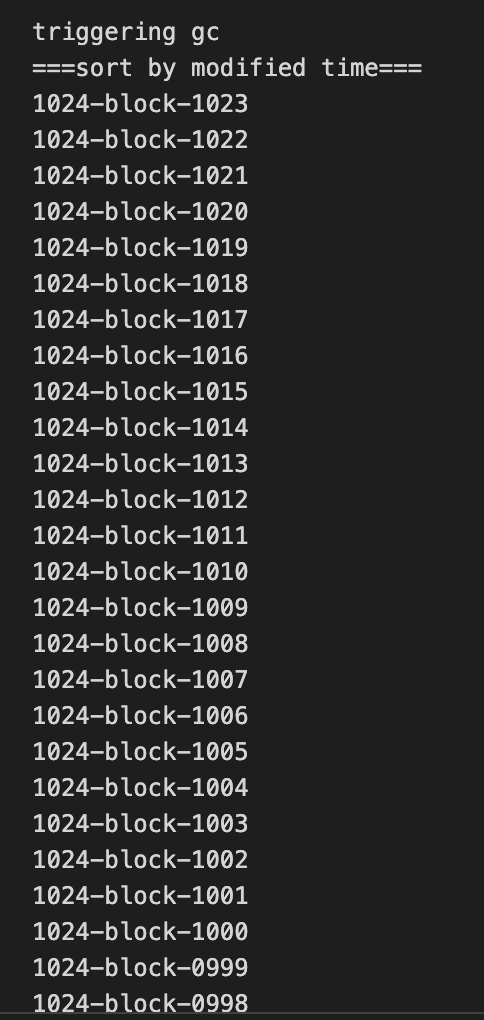
(too long, cannot be screen-captured)





my\_3.txt is my output, test3\_out.txt is the standard output

**Task 4**



(too long, cannot be screen-captured)



* 1. **Output of Bonus**

Haven’t been finished, but already has initial thoughts that are written in my codes.

**--- End of Report ---**