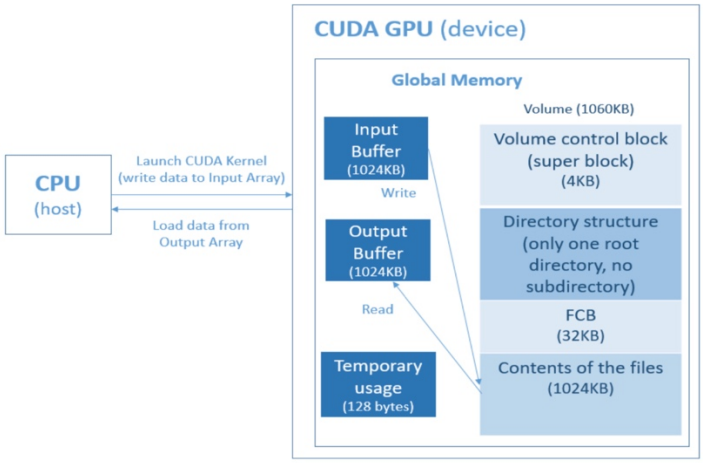
Report for CSC3150 Assignment 3

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 1024bytes 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 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 is set to 0 byte, as required, we still allocate a total 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 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 interpret the FCB index information from the file pointer that passes in as a parameter, use this to get its FCB block, 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 block 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 is originally some contents in the target file (this information can again be retrieved from the FCB block of this file), we need to remove it. Apart from removing it, for later implementation, I modify the bitmap such that all the files except its start file 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 contains 1 storage block that belongs to it. If the size is greater than 32, we first need to go through the bitmap, from the storage block that belongs to it, and check how many empty storage blocks available after its "start" block. If the number of available 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, 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, 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. We search the

bitmap to look for the first 0 bit, which is the first empty space of it, then this will become the start index of the storage blocks that stores the contents of the target file, 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 it, 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 block 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 file 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. 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.

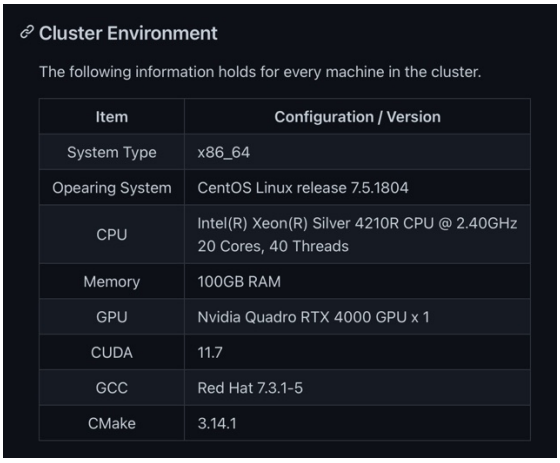
In bonus, I mainly reimplement the function of g*sys(fs, LS_D, LS_S)*, and *fs_gsys(fs, MKDIR, “app/0”)*. I haven’t finished the implementation of *fs_gsys(fs, CD, “app/0”)*. Now they are able to be compiled, please refer to my code as a reference!

2. Program Execution Environment

2.1 OS

2.2 VS version

2.3 CUDA Version and GPU information (Cluster Environment)



3. 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.

```
[120090874@node21 ~]$ cd /nfsmnt/120090874/A3-template
[120090874@node21 A3-template]$ sbatch ./slurm.sh
```

4. 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…

5. Problems Met in this Assignment

(1) Understanding the concepts

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

(2) Debugging issues

1. 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)
2. 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).
3. 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 to 0, then set 0 to 1), otherwise, problems may occur.
4. We must initialize the contents in the volume array before writing anything to it, like

```
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 waste me a lot of time, I don't know why…

6. Screenshots of the Program Outputs

Task 1

```
===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
```

```
[120090874@node21 A4-template]$ cmp -i 32:0 data.bin snapshot.bin
data.bin snapshot.bin differ: byte 33, line 1
[120090874@node21 A4-template]$
```

Task 2

```
[120090874@node21 A4-template]$ cmp -i 32:0 data.bin snapshot.bin
data.bin snapshot.bin differ: byte 33, line 1
[120090874@node21 A4-template]$
```

```
===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
```

Task 3

```
===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
{ABCDEFGHIJKLM 1021
zABCDEFGHIJKLM 1020
```

(too long, cannot be screen captured)

```
[120090874@node21 A4-template]$ cmp -i 1000:1000 data.bin snapshot.bin
data.bin snapshot.bin differ: byte 1025, line 4
[120090874@node21 A4-template]$ cmp -i 32:0 data.bin snapshot.bin
data.bin snapshot.bin differ: byte 33, line 1
```

```
[120090874@node21 A4-template]$ cmp test3_out.txt my_3.txt
```

my_3.txt is my output, test3_out.txt is the standard output

Task 4

```
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
```

(too long, cannot be screen captured)

6.3 Output of Bonus

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

--- End of Report ---