

Operating Systems

File System

11/28/2022

CONTENTS

1. Development environment	3
2. Execution Steps	4
3. Program Design	5
3.1 Theoretical Background	
3.2 Program Implementation	
4. Program Output	13
5. Encountered Problems & Solution	17
6. Learning outcome	18

1. Development environment

For this assignment, I used cluster **Slurm** provided by CSC4005 to write and test my program. Specifically, there is an overview given by the CSC4005 TAs:

Slurm is an open source, fault-tolerant, and highly scalable system that we choose for cluster management and job scheduling. As a cluster workload manager, Slurm has three key functions.

- 1. It allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time so they can perform work.*
- 2. It provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes.*
- 3. It arbitrates contention for resources by managing a queue of pending work.*

The **working system configurations** are described below:

Item	Configuration / Version
System Type	x86_64
Operating System	CentOS Linux release 7.5.1804
CPU	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz 20 Cores, 40 Threads
Memory	100GB RAM
GPU	Nvidia Quadro RTX 4000 GPU x 1
CUDA	11.7
GCC	Red Hat 7.3.1-5
CMake	3.14.1

Figure 1: System Config

The host machine (the physical machine I used to connect to the slurm) configurations are described below:

- System Type: x64-based PC
- Operating System: Windows 11 10.0.22621
- CPU: Intel(R) Core (TM) i7-10750H CPU @ 2.60GHz
- Memory: 16GB
- GPU: NVIDIA GeForce RTX 2060
- GCC: 12.2.0
- CMake: 3.23.0

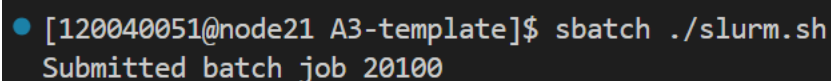
In order to establish a connection between my PC and the cluster, I used **Romote SSH** extension in the VS Code.

2. Execution Steps

In this assignment, the program is submitted to the cluster for execution using **sbatch** with the provided batch script. Specifically:

sbatch is used to submit a job script for later execution. The script will typically contain one or more srun commands to launch parallel tasks. If you use sbatch for job submission, you will get all the output recorded in the file that you assign in the batch script.

The corresponding command line is: `sbatch ./slurm.sh`



```
• [120040051@node21 A3-template]$ sbatch ./slurm.sh
Submitted batch job 20100
```

Figure 2: use sbatch to submit

In addition, the program can also run in an interactive way using self-defined **salloc** and **srun** commands. Specifically:

salloc is used to allocate resources for a job in real time. Typically, this is used to allocate resources and spawn a shell. The shell is then used to execute **srun** commands to launch parallel tasks.

srun is used to submit a job for execution in real time.

srun hostname prints the hostname of the nodes allocated.

For simplicity, during the writing and testing, I only adopt the method using **sbatch** with the given batch script.

3. Program Design

3.1 Theoretical Background

In this project, the target is to implement the mechanism of file system management via GPU's memory. The project takes the **global memory as a volume** (logical drive) from a hard disk, and extract the information from the volume **by single thread**.

The basic program structure is illustrated below:

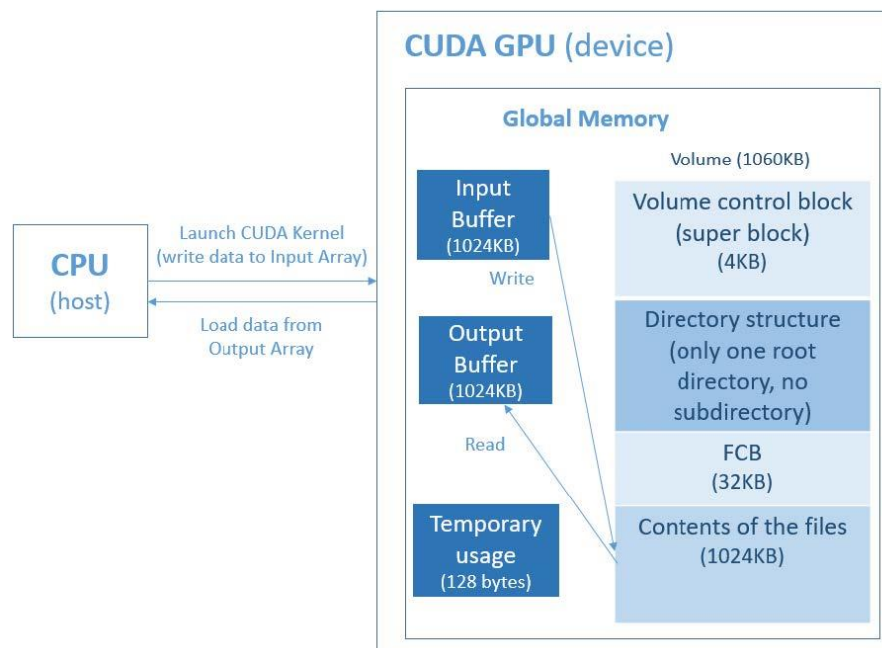


Figure 3: Program Structure

Program Configurations:

- Volume size (hard disk): 1060 KB (1085440 bytes):
 - Volume Control Block (VCB): 4 KB (4096 bytes)
 - ◆ 32 K bits
 - File Control Block (FCB): 32 KB
 - ◆ Each FCB size: 32 bytes
 - ◆ FCB Entries: 1024
 - Total Files: 1024 KB (1048576 bytes)
 - ◆ Maximum file name size: 20 bytes
 - ◆ Maximum file number: 1024
 - ◆ Files are stored in contiguous blocks
 - ◆ Block size: 32 bytes; Block number: 32 K
- I/O Buffer: 1024 KB
- Extra Space: 128 bytes

With the configurations mentioned above and the provided assignment template, there are mainly 5 functions (or operations) that needs to be designed and implemented by ourselves, namely: **fs_open fs_read fs_write fs_gsys(RM) fs_gsys(LS_D/LS_S)** Next, I will explain the general mechanism of my designed File System based on the above operations.

fs_open(G_Read) will search through the file name part within each FCB for the provided filename and **return the found FCB index** (0-1023); If not found, there will be an **assertion error**.

fs_open(G_Write) will search through the file name part within each FCB for the provided filename and **return the found FCB index** (0-1023); If not found, the function will **request an empty block** from the VCB (if there's no empty block, the function will print out the error information and **return 0 immediately**). Then the function will search for **an empty FCB entry** (If not found, there will be an **assertion error**) and set the meta data (file size is set to 0) for this file and **return this FCB index**.

fs_read will access the file's starting block index information in the FCB with the provided file pointer (FCB index), and read the specified size of file data into the output buffer.

fs_write will access the file size information in the FCB with the provided file pointer (FCB index) and transform it to **the number of blocks covered (specified as file length)** and do the same for the writing size. Then the function will go to different branches based on the comparison of file length and write length.

1. **Write length == file length**, which is the simplest situation. The function will continue to access the file's starting block index information in the FCB and write the target value to the storage, and update the metadata (filesize, modified time) in the FCB and return the file pointer. **The VCB is unchanged, and no compaction is involved.**
2. **Write length != file length**. Under the circumstance, the function will first check **whether there enough blocks**, including the blocks taken by the current file, in the storage (if no, the function will print out the error information and return the pointer immediately).

Next the function will get the **first empty block (0 bit)** in the VCB (The compaction algorithm ensures that the files are written contiguously and there should be no gap between the files) and check **whether it is next to where the current file ends**. And based on the comparison result, the function will further split into two streams.

a) The first empty block is next to where the current file ends

In this situation, we still don't need to do the compaction, since the current file is already the last file among the compacted files. The function will continue to remove file contents and write the target data into the storage **from the same start index**. Then the function will update the VCB and FCB correspondingly.

b) The first empty block (storage end) is NOT next to where the current file ends. In this situation, unfortunately, we have to do the **compaction**. However, the compaction logic is not difficult:

1. Move all the **contents** right after the current file up to the storage end forward till where the current file begins;
2. Set the threshold as the block index next to the current file ends. For every FCB entry, **check if the recorded starting block index is greater or equal to the threshold**, if yes, then **minus that index with the file length**;
3. Write the file data to the storage starting from the **storage end – file length** (see figure 5)
4. Set the VCB according to the difference between the file length and the write length (erase 1s or add 1s); Set the corresponding FCB information for this file.

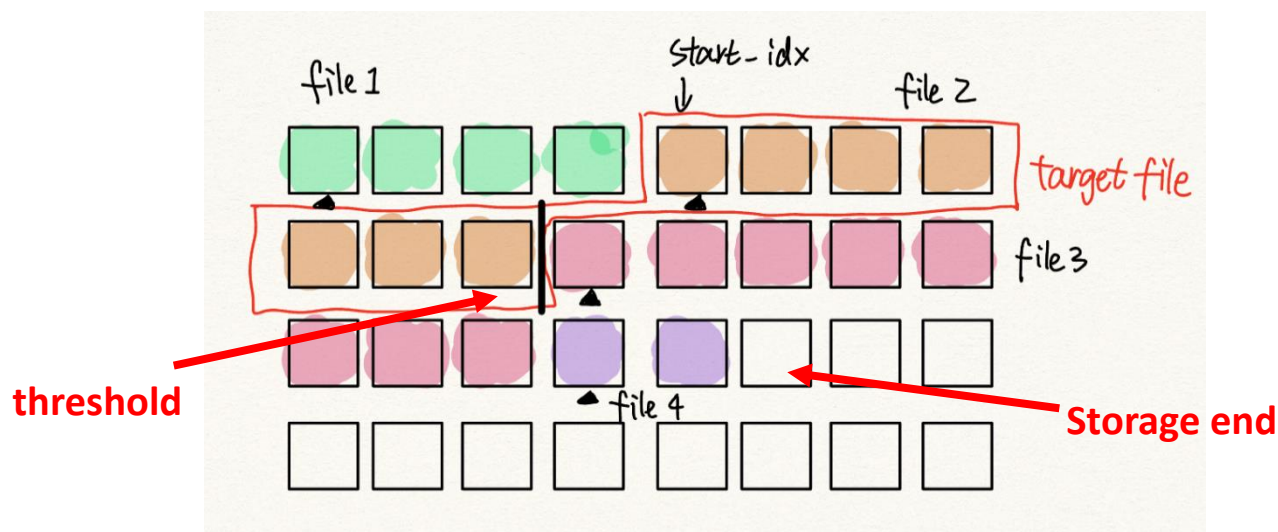


Figure 4: Before Compaction

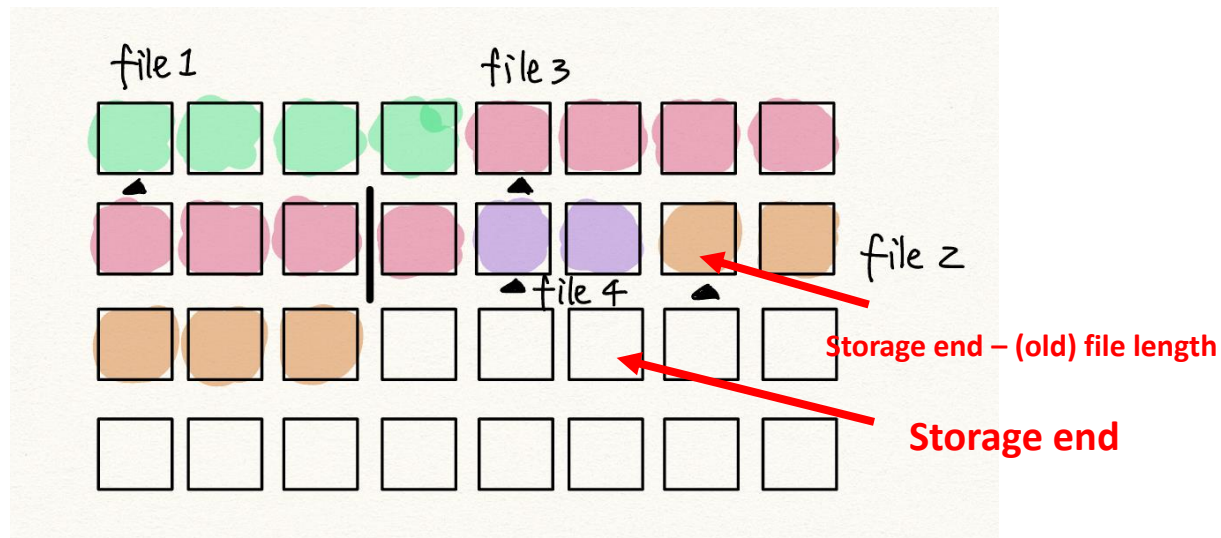


Figure 5: After Compaction

fs_gsys(RM) will first find the target FCB with the provided filename (If not found, the function will print out the error message and return). Then remove the FCB entry, file content, and reset the VCB according to the start index and the file length.

fs_gsys(LS_D) will create **two temporary arrays** with **MAX_FILE_NUM** of elements for **file ids** (FCB indices) and **last modified time**. Then the function will apply **insertion sort** based on the **last modified times**, with the file ids array following the exact operations to **stay matched with the last modified time**. After sorting, the function will print out the filenames following the sequence of the most recent modified to the least recent modified.

fs_gsys(LS_s) will create **three temporary arrays** with **MAX_FILE_NUM** of elements for **file ids** (FCB indices) and **created time** and **file sizes**. Then the function will apply **insertion sort** based on the file sizes, with the file ids and create time array following the exact operations to **stay matched with the file sizes**. After sorting, the function will print out the filenames following the sequence of the largest file to the smallest file.

3.2 Program Implementation

In this section, I will demonstrate some key implementation logic for this assignment.

```

/*
    volume 1 byte per entry

    [VCB]: volume[0:4095], 4KB, 32K places, 1 place (1 bit) per block;
    | | | | For each block: 0 is empty, 1 is full.

    [FCB]: volume[4096:36863], 32KB, 1024K places, 1 place (32B) per file
    | | | | In each place:
    | | | | Index      Size      Valid Range      Item
    | | | | 0-19      20 B      NA                Filename
    | | | | 20-23      2 B      0~4K*1024KB-1      File Size (0-1024KB)
    | | | | 24-25      4 B      0~64K-1            File starting block index
    | | | | 26-27      2 B      0~64K-1            Created Time
    | | | | 28-29      2 B      0~64K-1            Modified Time
    | | | | 30         1 B
    | | | | 31         1 B      0~256              Valid Bit (0/1)

    [File]: volume[36864:1085439], 1024KB, 32K blocks, 1 block (32B) per file
*/

```

Figure 6: FCB Setting

```

// init every volume entry to 0
for (int i=0; i < fs->STORAGE_SIZE; i++){
    fs->volume[i] = 0;
}

```

Figure 7: Initiate volume to 0

Helper function #1: set VCB to target value at the specified start with specified length

```

/* ----- Some helper functions are defined below ----- */

// erase the corresponding bits to 0 in VCB
// mode = 0: erase to 0; mode = 1, set to 1
__device__ void set_file_bits(FileSystem *fs, u16 start, u16 b_len, int mode) {

```

Figure 8

Helper function #2: count the number of zeros in VCB

```
// Count number of zeros in FCB
__device__ int count_zeros(FileSystem *fs) {
```

Figure 9

Helper function #3: get the first 0 bit from the VCB starting from index 0.

```
// bits operation helper function
// flag = 1, assign, flag = 0, do not assign
__device__ u16 get_0_bit_in_VCB(FileSystem *fs, int set){
```

Helper function #4 - #6: self-defined strcpy, strcmp, strlen.

Pointer type casting.

```
// copy file name
_strcpy((char *) &(fs->volume[FCB_base]), filename);
// set file size (0 byte)
*((u32 *) &(fs->volume[FCB_base+20])) = 0;
// set block index
*((u16 *) &(fs->volume[FCB_base+24])) = ret_index;
// set create time
fs->volume[FCB_base+26] = gtime / 256;
fs->volume[FCB_base+27] = gtime % 256;
// set modified time
fs->volume[FCB_base+28] = gtime / 256;
fs->volume[FCB_base+29] = gtime % 256;
gtime++;
// set valid bit
fs->volume[FCB_base+31] = 1;
```

Figure 10: set FCB

Ceiling

```
if (file_size == 0)
{
    file_len = 1;
}
else
{
    file_len = (file_size + fs->STORAGE_BLOCK_SIZE - 1) / fs->STORAGE_BLOCK_SIZE;
}
```

Figure 11: Simple ceiling arithmetic

Compaction implementation

```
// compaction
// move the below contents upwards
for (int i = next_start; i < storge_end; i++)
{
    fs->volume[i - gap] = fs->volume[i];
}

// change FCB, should not involve this fp
for (int i = 0; i < fs->FCB_ENTRIES; i++)
{
    u32 fcb_base_ = fs->SUPERBLOCK_SIZE + i * fs->FCB_SIZE;
    if (fs->volume[fcb_base_ + 31] != 0)
    {
        u16 fb_idx = *((u16 *)&(fs->volume[fcb_base_ + 24])); // fetch block index in FCB
        if (fb_idx >= next_index) // in the moved batch
        {
            *((u16 *)&(fs->volume[fcb_base_ + 24])) = fb_idx - file_len; // moving offset
        }
    }
}

// change VCB
if (write_len > file_len)
{
    set_file_bits(fs, end_block, (write_len - file_len), 1); // add bits
}
else if (write_len < file_len)
{
    set_file_bits(fs, (end_block - file_len + write_len), (file_len - write_len), 0); // erase bits
}
```

Figure 12: Compaction

```
u16 write_idx = end_block - file_len;
u32 storge_write_start = fs->FILE_BASE_ADDRESS + write_idx * fs->STORAGE_BLOCK_SIZE;

// clear the remainings
for (int i = 0; i < gap; i++)
{
    fs->volume[storge_write_start + i] = 0;
}

// write value
for (int i = 0; i < size; i++)
{
    fs->volume[storge_write_start + i] = input[i];
}

// change FCB for this fp
*((u32 *)&(fs->volume[FCB_base + 20])) = size;
*((u16 *)&(fs->volume[FCB_base + 24])) = write_idx;
fs->volume[FCB_base + 28] = gtime / 256; // modified time
fs->volume[FCB_base + 29] = gtime % 256;
gtime++;
```

Figure 13: Compaction Cont'd

Insertion sort (two arrays)

```
printf("===sort by modified time===\n");
// insertion sort
for (int i = 1; i < 1024; i++)
{
    int key = f_time[i];
    u16 idx = f_idx[i];
    int j = i - 1;

    while (j >= 0 && key > f_time[j])
    {
        f_time[j + 1] = f_time[j];
        f_idx[j + 1] = f_idx[j];
        j = j - 1;
    }

    f_time[j + 1] = key;
    f_idx[j + 1] = idx;
}
```

Figure 14: Insertion sort

4. Program Output

The program successfully passed all the 4 provided test cases.

Test case 1

<pre>===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</pre>	<pre>===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</pre>
---	---

Figure 15: test case 1 results VS demo output

Test case 2:

```

===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===
*ABCEFGHIJKLMNOPQR 33
)ABCEFGHIJKLMNOPQR 32
(ABCEFGHIJKLMNOPQR 31
'ABCEFGHIJKLMNOPQR 30
&ABCEFGHIJKLMNOPQR 29
%ABCEFGHIJKLMNOPQR 28
$ABCEFGHIJKLMNOPQR 27
#ABCEFGHIJKLMNOPQR 26
"ABCEFGHIJKLMNOPQR 25
!ABCEFGHIJKLMNOPQR 24
b.txt 12
===sort by modified time===
*ABCEFGHIJKLMNOPQR
)ABCEFGHIJKLMNOPQR
(ABCEFGHIJKLMNOPQR
'ABCEFGHIJKLMNOPQR
&ABCEFGHIJKLMNOPQR
b.txt

===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===
*ABCEFGHIJKLMNOPQR 33
)ABCEFGHIJKLMNOPQR 32
(ABCEFGHIJKLMNOPQR 31
'ABCEFGHIJKLMNOPQR 30
&ABCEFGHIJKLMNOPQR 29
%ABCEFGHIJKLMNOPQR 28
$ABCEFGHIJKLMNOPQR 27
#ABCEFGHIJKLMNOPQR 26
"ABCEFGHIJKLMNOPQR 25
!ABCEFGHIJKLMNOPQR 24
b.txt 12
===sort by modified time===
*ABCEFGHIJKLMNOPQR
)ABCEFGHIJKLMNOPQR
(ABCEFGHIJKLMNOPQR
'ABCEFGHIJKLMNOPQR
&ABCEFGHIJKLMNOPQR
b.txt

```

Figure 16: test case 2 output VS demo output

Test case 3

<pre> ===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 </pre>	<pre> ===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 </pre>
---	---

Figure 17: test case 3 output VS demo output

```

C:\Users\Xiaoyuan\Desktop\1>fc result.out test3_out.txt
正在比较文件 result.out 和 TEST3_OUT.TXT
FC: 找不到差异

```

Figure 18

Test case 4

```

C:\Users\Xiaoyuan\Desktop\1>fc snapshot.bin data.bin
正在比较文件 snapshot.bin 和 DATA.BIN
FC: 找不到差异

```

Figure 19

<pre> 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 </pre>	<pre> 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-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 </pre>
--	--

Figure 20: test case 4 output VS demo output

```

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

```

Figure 21: test case 4 cont'd

5. Encountered Problems & Solution

During the debugging, I encountered a problem about pointer type casting. When I implemented type casting to the pointers for storing and extracting value in the array as a whole, casting (uchar*) into (char*) and (u16*) is fine; however, casting (uchar*) into (int*) or (u32*) will cause the program to crash implicitly (i.e., no error message was shown but the program exits).

In the following picture, using the second expression will cause the program to fail.

```
// set file size (0 byte)
*((u16 *) &(fs->volume[FCB_base+22])) = 0;
*((u32 *) &(fs->volume[FCB_base+22])) = 0;
// printf("(if) set file_size successs, fcb base = %d\n", FCB_base);
```

Figure 22: wrong pointer type casting

However, after I discussed with the TA and looked it up on the Internet. I realized that this problem is triggered by the **address alignment requirement**. Specifically:

What does it mean to be 4-byte aligned?

A 1-byte variable (typically a char in C/C++) is always aligned. A 2-byte variable (typically a short in C/C++) in order to be aligned must lie at an address divisible by 2. A 4-byte variable (typically an int in C/C++) must lie at an address divisible by 4 and so on.09-Apr-2020

Figure 23: [source](#)

In the above picture, the address of FCB+22 is not divisible by 4, so the type casting is erroneous. After I changed it to FCB + 20, then it went well.

6. Learning Outcome

Through this assignment, I've learned to:

1. Design VCB, FCB and manage a file system structure.
2. Implement compaction algorithm in the file system with less time cost.
3. Deal with bit operations (masking, shifting, toggling) in VCB.
4. Do pointer type casting and address alignment requirement.
5. Review and implement the sort algorithm.
6. Understand the working mechanism of the file system as a whole.