CSC3150 Project 4 Report

119010341 WU Yifan

Running Environment

- Windows 10
- NVIDIA CUDA 11.2.162 driver
- NVCC version:

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Mon_Nov_30_19:15:10_Pacific_Standard_Time_2020
Cuda compilation tools, release 11.2, V11.2.67
Build cuda_11.2.r11.2/compiler.29373293_0
```

- NVIDIA Geforce GTX 1650
- Visual Studio 2019 (developing toolkit for c++/c)

Execution Guide

Go to the assignment directory

```
|
├─bonus
|      data.bin
|      main.cu
|      user_program.cu
|      file_system.cu
|      file_system.h
|      snapshots.bin
|      README.txt
|
└─source
        data.bin
        main.cu
        user_program.cu
        file_system.cu
        file_system.h
        snapshots.bin
        README.txt
```

Go to the basic task directory or the bonus directory

```
data.bin
main.cu
user_program.cu
file_system.cu
file_system.h
snapshots.bin
README.txt
```

Compilation

The script is included in the file `README.txt`

```
nvcc --relocatable-device-code=true main.cu file_system.cu user_program.cu -o
main.out
```

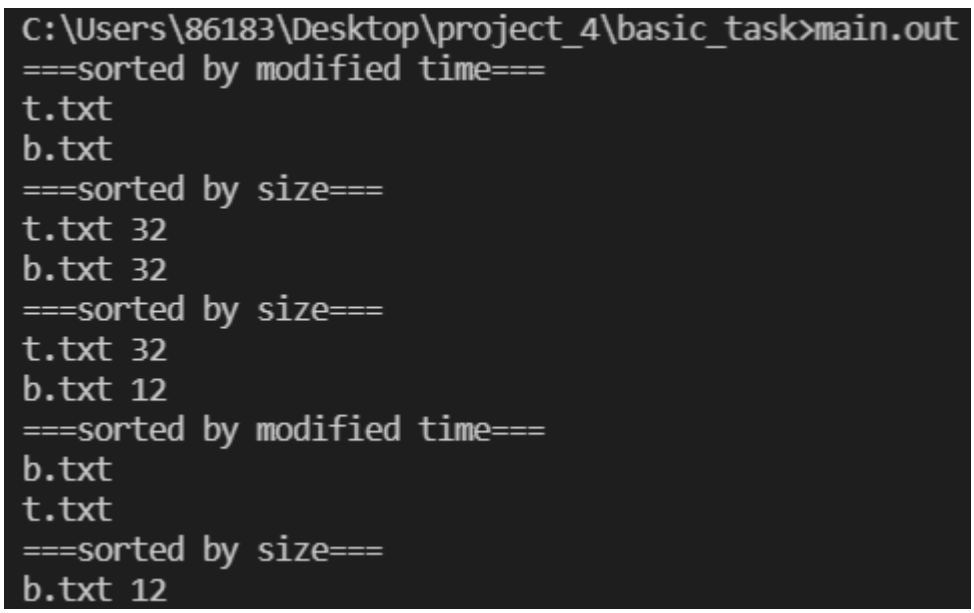
3 new files should appear, and then choose `main.out` to execute

```
main.exp
main.lib
main.out
```

Sample Output

Sample output for the basic task

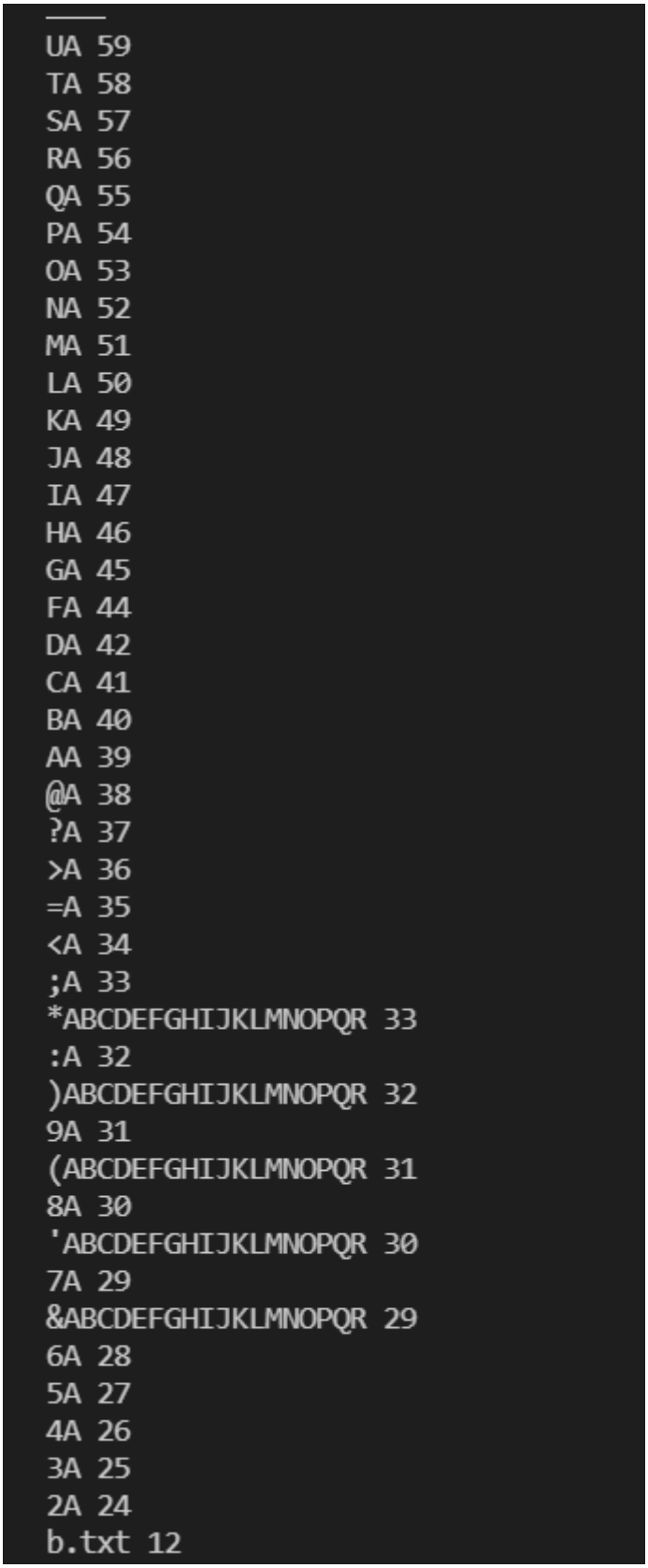
- case 1



- case 2

```
C:\Users\86183\Desktop\project_4\basic_task>main.out
===sorted by modified time===
t.txt
b.txt
===sorted by size===
t.txt 32
b.txt 32
===sorted by size===
t.txt 32
b.txt 12
===sorted by modified time===
b.txt
t.txt
===sorted by size===
b.txt 12
===sorted by size===
*ABCDEFGHIJKLMNOPQR 33
)ABCDEFGHIJKLMNOPQR 32
(ABCDEFGHIJKLMNOPQR 31
'ABCDEFGHIJKLMNOPQR 30
&ABCDEFGHIJKLMNOPQR 29
%ABCDEFGHIJKLMNOPQR 28
$ABCDEFGHIJKLMNOPQR 27
#ABCDEFGHIJKLMNOPQR 26
"ABCDEFGHIJKLMNOPQR 25
!ABCDEFGHIJKLMNOPQR 24
b.txt 12
===sorted by modified time===
*ABCDEFGHIJKLMNOPQR
)ABCDEFGHIJKLMNOPQR
(ABCDEFGHIJKLMNOPQR
'ABCDEFGHIJKLMNOPQR
&ABCDEFGHIJKLMNOPQR
b.txt
```

- case 3



Sample output for the bonus task

```
C:\Users\86183\Desktop\project_4\bonus>main.out
===sorted by modified time===
t.txt
b.txt
===sorted by size===
b.txt 32
t.txt 32
===sorted by modified time===
app d
t.txt
b.txt
===sorted by size===
b.txt 32
t.txt 32
app 0 d
===sorted by size===
===sorted by size===
a.txt 64
b.txt 32
soft 0 d
===sorted by modified time===
soft d
b.txt
a.txt
app/soft/
===sorted by size===
D.txt 1024
C.txt 1024
B.txt 1024
A.txt 64
===sorted by size===
a.txt 64
b.txt 32
soft 24 d
app/
===sorted by size===
b.txt 32
t.txt 32
app 17 d
===sorted by size===
a.txt 64
b.txt 32
===sorted by size===
b.txt 32
t.txt 32
app 12 d
```

Program Design--Basic Task

1. Task

In this task, we are going to implement the **file system** using CUDA (Compute Unified Device Architecture) in the GPU due to GPU's small size and low latency access.

Below are the requirements for the implementation

Paging System Parameters	Values
Volume Size	1060KB
File Size	<=1024KB
File Number	<=1024
FCB Size	32B
FCB Number	1024
Block Size	32B
Block Number	32768

The file system also needs to implement **contiguous allocation** and **file compaction**.

2. General Storage Allocation Design

There are mainly 2 ways to implement file allocation: Linked list or Bit map. Due to the limitation of volume size, it is better to use the bit map.

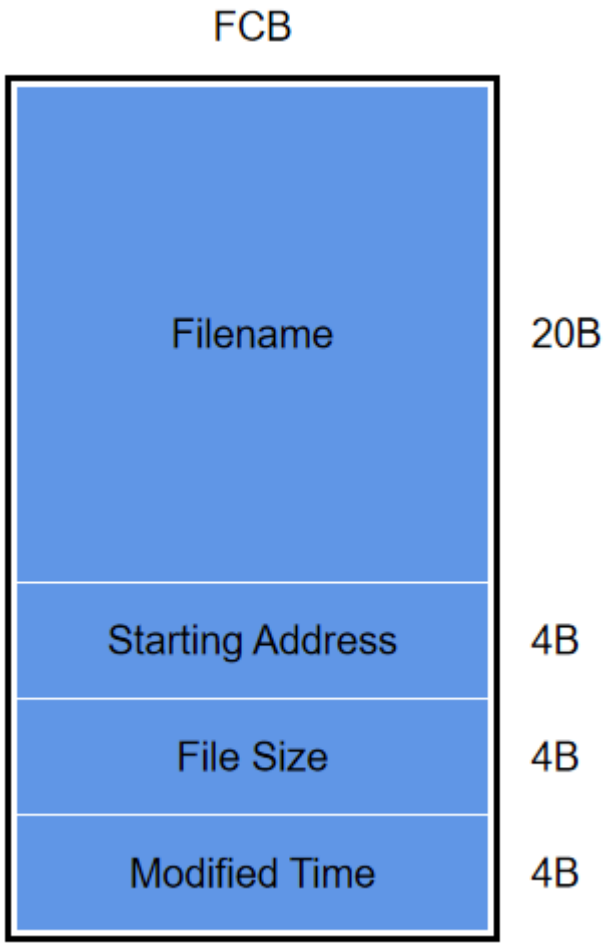
So we can divide the volume into 3 main parts:

- File contents ---- 1024B*1024=1024KB
- FCBs (File Control Blocks) ---- 32B*1024=32KB
- Bit map ---- 1060KB-32KB-1024KB=4KB

Bit map

There are in total 32768 (2^15) blocks for file contents, we only have 4KB=32768bit. Thus each block will take up 1 bit in the bit map.

FCB



3. fs_open

G_READ

In reading mode, we only need to search for the filename of the target file among FCBs, and return the FCB index as the read pointer.

While searching, for each non-empty FCB, we will compare each character of its filename with that of the target filename. The comparison terminates when either the current FCB or the target filename reads a `'\0'`

G_WRITE

In the write mode, we will also begin with searching the target file.

1. If the target file doesn't exist, we create a new file named as the target file.

   We will find the empty block with the smallest index using the bit map and find the empty FCB with the smallest index. The starting address, initial file size (0), and the modified time will be initialized. Then a write pointer (representing the empty FCB with the smallest index) will be returned.

2. If the target file already exists in the volume, we will first delete it by calling `fs_gsys(fs, RM, target_file)`. After that we will once again create a file named as that target file (repeat case 1)

Note that, to avoid too much bit operation, here we can use the type cast from a character pointer to an unsigned 32 bit integer pointer

```
// use type cast to set FCB data
starting_address = (u32)&fs->volume[fs->SUPERBLOCK_SIZE + fpfs->FCB_SIZE + fs->MAX_FILENAME_SIZE];
```

4. fs_read

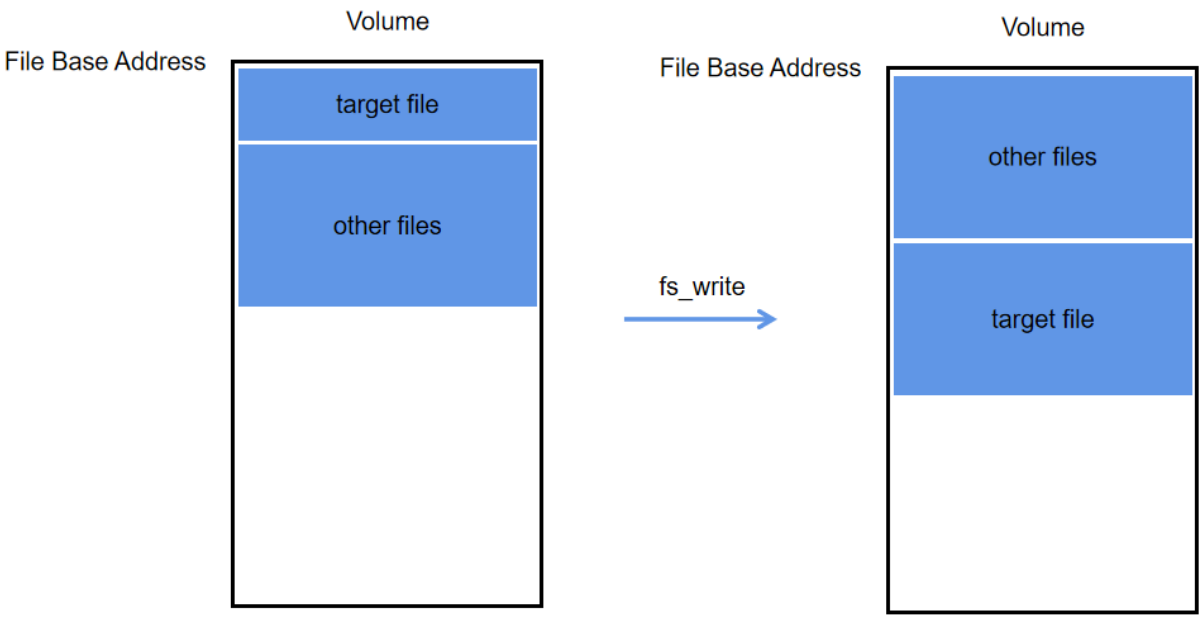
Read function is rather simple, since we have the read pointer (the corresponding FCB index), we can access the starting address of that file and read data of a specific size.

5. fs_write

We know that every time we open a file in the write mode, the original data should be cleared up before write new data into the file. However, we cannot ensure whether the size of new data will extend the old file size or not.

So here we design a mechanism to avoid size extension and at the same time use compaction to eliminate external fragmentations.

1. Each time the user invokes the `fs_write()`, the `fs_write()` function will reopen the target file in write mode.
2. `fs_open()` will call `fs_gsys(fs, RM, target_file)` to remove the original file and do the compaction
3. Then `fs_open()` will create the target file again. However, this time the target file will be placed at the backmost position among all the files, which prevents size extension.
4. Finally, `fs_write()` will perform file writing. New data will be written into the target file and the FCB as well as the bit map will be updated according to the properties (file size and modified time) of the target file



Note that, each time the user invokes `fs_write()`, the system clock `gtime` will increase by 1

6. fs_gsys (RM)

In `fs_gsys(RM)` we need to complete 2 tasks:

- clear up all the data related to the target file
- compaction

Since `fs_gsys(RM)` will receive the target file name as an argument, we first need to find the target file among FCBs. The searching mechanism is the same as it in `fs_open()`

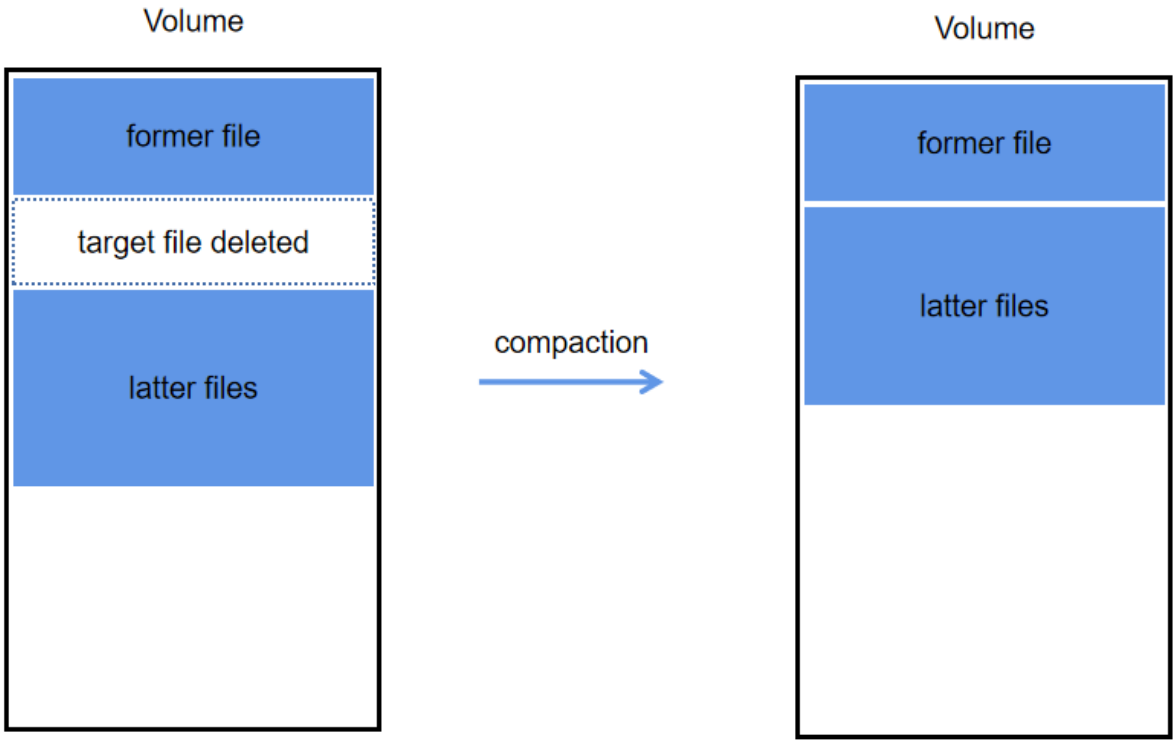
Then we will do data deleting:

- clear up FCB according to the target file FCB index
- clear up file contents according to the target file starting address and the file size
- update the bit map according to the target file starting address and the file size

After that, all the data related to the target file have been deleted. Then there may be a huge external fragmentation popping out between the files before the target file and the files after it. So we need to compact the data

There are mainly 2 steps for compaction:

1. Traverse the FCB entries as well as the bit map to find out the next non-empty FCB and non-empty block (indicating the next file)
2. Move the subsequent data in the FCBs, and the file content blocks forward (& update the bit map as well)



However, moving data in the FCBs is not enough. Since the file contents have been moved forward, the starting address of each file also changes. How can we update the starting address of each FCB?

We traverse the FCBs, if an FCB's starting address is bigger than the deleted file's starting address, it indicates that this file is a latter file. So we just adjust its starting address. We can simply minus its starting address by the difference between the starting address of the next file and the deleted file.

```
// for each FCB whose starting address is larger than that of the deleted file
starting_address -= (latter_files_starting_block - deleted_file_starting_block);
```

7. fs_gsys (LS_D / LS_S)

In the FCBs we have stored the file size as well as the modified time of each file, so we only need to perform a sorting algorithm to print the filenames in some order.

For `LS_D` or `LS_S`, we will create an array to hold the modified time or the file size of each FCB

```
// example
u32 size_array[1024]={0};
```

There are totally 1024 entries which corresponds to the FCB entries.

We will traverse the FCBs, and for each non-empty FCB we will record its file size/modified time into our array. In the meanwhile we will count the number of non-empty FCB (number of files).

Assume that there are `n` files in total. After that, we will

1. traverse the array to find out the maximum modified time/file size
2. print out the filename of that file
3. reset the corresponding entry in our array to be 0
4. *go back to step 1 (loop for n times)*

Program Design--Bonus

1. Task

Based on the basic task, we need to implement **tree-structured directories**

2. General Storage Allocation Design

In this task, we add some slight change to the volume storage:

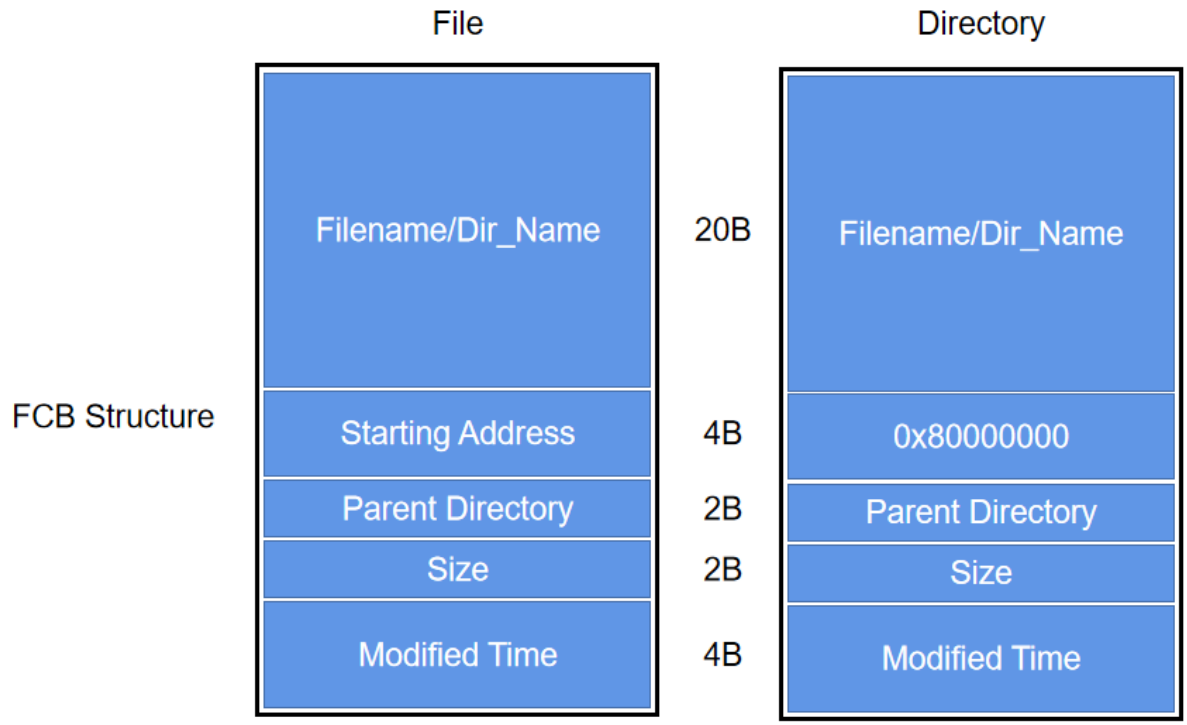
Paging System Parameters	Values
Volume Size	1085440B+32B+4B = 1085476B
FCB Entry Number	1025

A directory is simply another file. So we can simply record a directory by writing data into an FCB.

Thus, we add another FCB representing the root directory (the 1025th FCB)

Also, we will need to identify which directory we are currently at. So we use another 4B to hold a directory pointer.

For each file or directory, we must record which directory it is in, thus we may need to change the structure of FCB.



We may need to determine whether an FCB represents a file or a directory. Thus we can set all the "Starting Address" to be 0x80000000. Because directories don't have any real starting address, and the starting address of any file will not exceed 32768 which is 0x00008000. By compare the starting address of an FCB with 0x80000000, we can know it represents a file or a directory.

3. For Old Functions

The only change we need to add to these functions which we have implemented in the basic task is that

- We need to compare the parent directory of the file we open with the current directory

Since we can only operate on those files which are in the current directory.

4. fs_gsys(fs, MKDIR, “app\0”)

Exactly the same with `fs_open(G_WRITE)`, except that we are dealing with a directory so we will always compare the starting address of an FCB with 0x80000000.

5. fs_gsys(fs, CD, “app\0”)

1. traverse the FCBs to find the target directory
2. change the value of the "current directory pointer" to the target directory's FCB index

6. fs_gsys(fs, CD_P)

We simply change the value of the "current directory pointer" to FCB index of its the parent directory.

7. fs_gsys(fs, RM_RF, “app\0”)

This part is very similar to the `fs_gsys(RM)`, except that we also need to delete subdirectories

Thus we can use a recursive way to clear up directories:

1. use `fs_gsys(RM)` to delete all the inner files in the current directory
2. perform compaction regarding the file contents (*see basic task*)
3. recursively invoke `fs_gsys(fs, RM_RF, subdirectory)` to clear up the subdirectories

This recursion method may be limited by the size of our computer stack memory, so in the main function I adjust the stack size to 32768 by

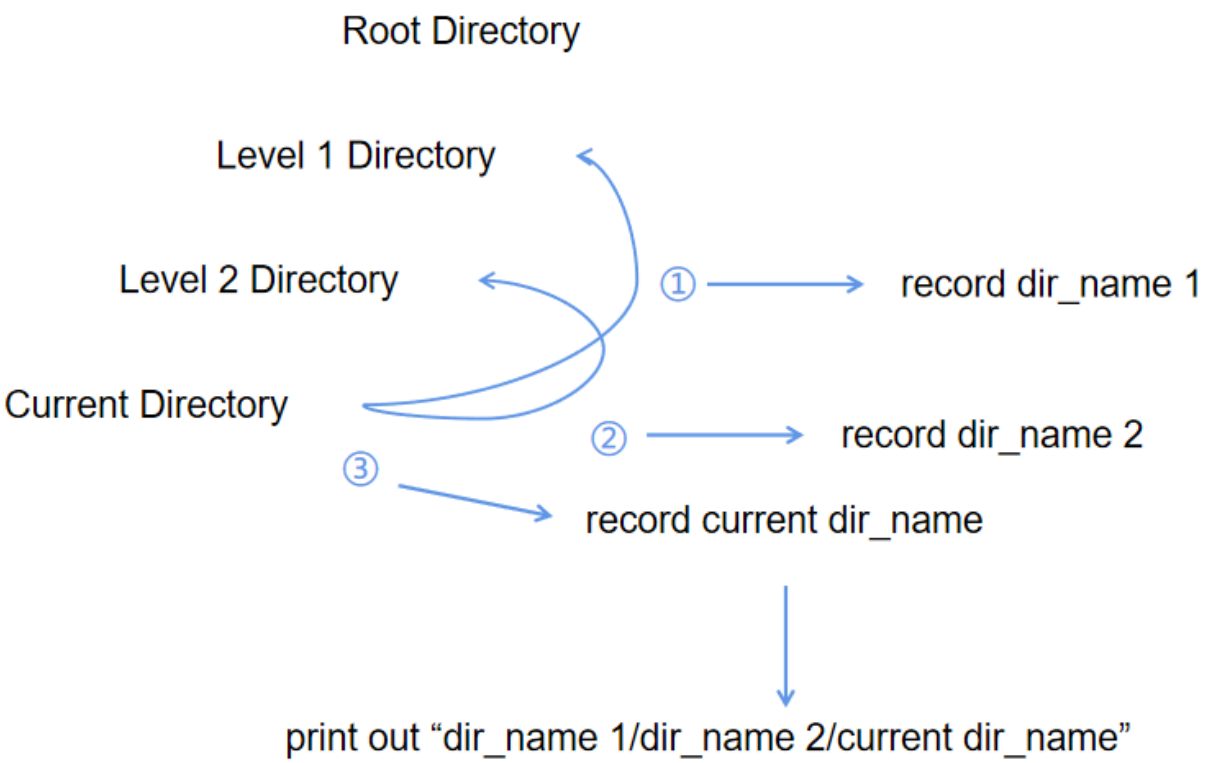
```
cudaDeviceSetLimit(cudaLimitStackSize, 32768);
```

8. fs, gsys(fs, PWD)

This function is relatively complicated since we only know the current directory but we need to print out beginning at the root directory. This means that we need to trace back to the foremost parent directory.

I used 2 levels of while loop together with a character array `char path[100];`,

- in each inner loop, we will trace back to the foremost parent directory and record its directory name into `path`
- this "foremost parent directory" should be replaced by its next level each outer loop



What I have learned

Through this assignment, I have learned about the structure as well as the mechanism of the file system which we use every day. It is hard to express the feeling when I find out that our file system can be implemented by some basic c++ codes. Thanks to this assignment I am more familiar with the contiguous allocation and the usage of bit map, and I have find my own way to implement the compaction of file contents. I realize that it can be really serious if the external fragmentations cannot be solved in time.