CSC3130 Assignment4 Report

Name: Hu Yiyan

Student ID: 119010104

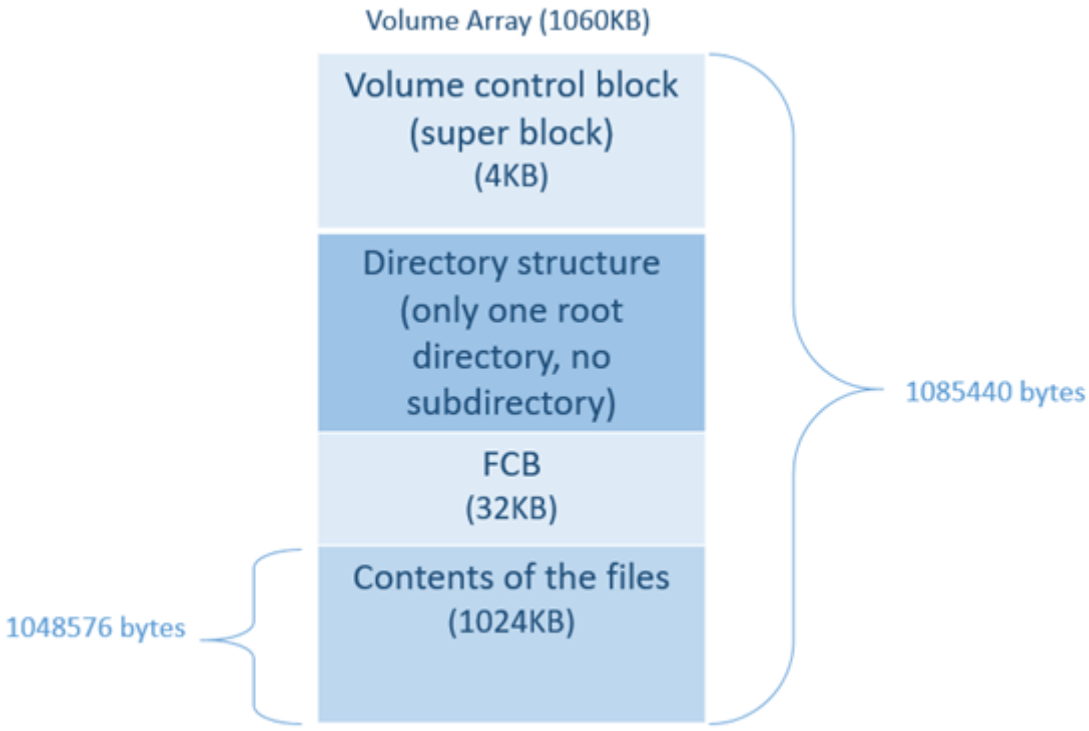
Date: November 24, 2021

Design

Design: Overview

1. File system structure

In the Assignment 4, we are required to implement a file system which supports some file operations. The file structure contains the logical storage unit and collection of the related information. The overall structure of the file system is shown below.



- The Volume control block (superblock) contains volume details including: total number of blocks in the file system, number of the free blocks, block size and free block array. In this assignment, we only need to maintain the free block array since the number of blocks is fixed and the block size is fixed.
- The File Control Block (FCB) contains details about the file including: inode number, permissions, size, dates and etc. In this assignment, to simply the task, we do not need to implement the permission in the FCB.
- Storage stores the content of the file. To efficiently utilize the storage space, we may implement algorithms to arrange file content allocation (eg: Compaction).

In this assignment, the memory distribution is fixed. The size of elements in the filesystem is shown below.

```
#define SUPERBLOCK_SIZE 4096 //32K/8 bits = 4 K
#define FCB_SIZE 32 //32 bytes per FCB
#define FCB_ENTRIES 1024
#define VOLUME_SIZE 1085440 //4096+32768+1048576
#define STORAGE_BLOCK_SIZE 32

#define MAX_FILENAME_SIZE 20
#define MAX_FILE_NUM 1024
#define MAX_FILE_SIZE 1048576

#define FILE_BASE_ADDRESS 36864 //4096+32768
```

In the following paragraphs, I will explain my the details of my design of the file system.

2. Superblock

In my design, I use the Superblock to maintain a free block array.

```
\because The superblock_size = 4096 bytes = 32768 bits
    While: the num_of_block_in_storage = storage_size / storage_block_size
                                       = 32768
\therefore Each bit of the superblock maps to a block in storage
\therefore We can implement a bit map in the superblock,
    where: bit[i] of superblock maps to block[i] of storage
```

If block **i** of the storage is occupied by file content, then the bit **i** of the superblock is **1**,

else if block **i** of the storage is free, then the bit **i** of the superblock is **0**.

3. FCB Entries

The FCB entry in this assignment has 32 bytes, we can use the 32 bytes to store the information about a file. I store the following information in a FCB entry of a file: (a) the name of the file, (b) the address of the file content in the storage, (c) the create time and modified of the file, (d) the size of file, and (e) the dirty bit

The detail of each information in the FCB entry:

```
// the FCB entry has 32 bytes
Name: byte[19:0] -> The file name;
// max size of file name is 20 byte
Fpointer: byte[21:20] -> The address of file content in storage;
// the max address is 0x7F, which is within the limit of 2 bytes
Time: byte[25:22] -> The modified time of the file;
CreateTime: byte[29:26] -> The created time of the file;
// time is stored in a u32 int
Size: byte[31:30] -> The size of the file;
// max size is 1024 (0x04_00), which is within the limit 2 bytes
DirtyBit: byte[21], bit[7] -> The dirty bit of the FCB entry:
// for the max address is 0x7F, there is one bit left in byte[21] for dirty bit
// dirty bit: 1->FCB entry is used by a file / 0->the FCB entry is free
```

The FCB structure:

4. Storage

The storage part of the file system stores the file content. The storage blocks have the following attributes: (1) The starting address of the file content space of a file is indicated by the **Fpointer** field of the FCB entry of the file, (2) Each block of the storage corresponds to a bit in bit map of the superblock.

To use the storage space more efficiently, we need to do the compaction to make the storage space compacted (with no external fragmentation). I implement a simple algorithm to maintain the compaction of the storage space. I do compaction every time I remove a file. The algorithm of the compaction is illustrated below

```
 when we need to remove a file:
    1. find the starting address of the file in the storage
    2. find the size of the file (unit: block)
    3. clear the file content in the storage according to the address and the
size of the file
    4. move all the file contents of the files after the removed file forward,
and fill the empty space of the removed file in the storage
    5. update the bit map in the superblock
```

The following figures further explain the algorithm:

Storage before compaction:

Storage after compaction:

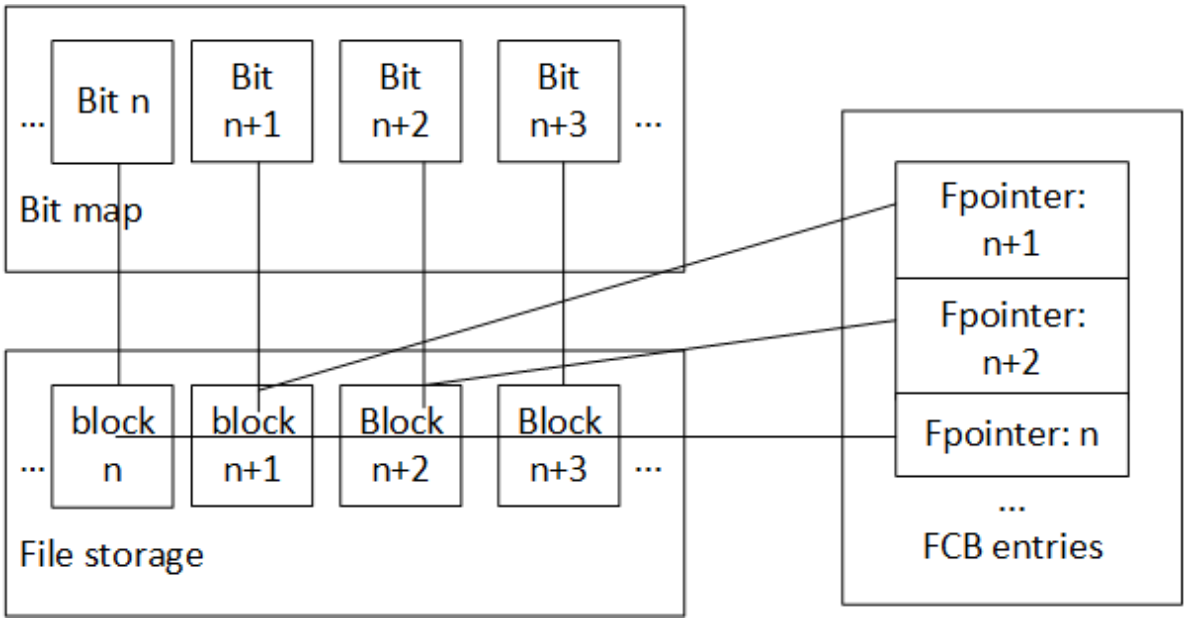
For we do the compaction every time we remove a file or write a file, then we can ensure that the file contents in the storage in the storage are always being compacted. The first 0 bit in the bit map will map to the beginning the free space in the storage.

Design: Source Part

1. Filesystem structure: FCB Entries, Superblock and Storage

In the source part of the Assignment 4, the structures of FCB entries, superblock and storage is exactly the same as the structures specified in the **Overview** part.

In this part, I will demonstrate the logical relationships between each block of the file system. The logical relationships of different blocks of the file system is shown below.



2. File Operations

We need to implement some file operations to allow users operate on files.

- fs_open

```
u32 fs_open(Filesystem *fs, char *s,  int op);
// return value: FCB entry index of the file
```

The operation opens a file, which follows the following logic:

```
If the operation is read
1. Try to find the FCB entry of the file acccording to the name
2. If the FCB entry is found, find the storage of the file and  return
the FCB entry index of the file.
   If the entry is not found, report an error.

If the operation is write
1. Try to find the FCB entry of the file according to the name
2. If the FCB entry is found, (a) remove the file content in the
storage, (b) find free space to reallocate the file content, (c) update
bit map in superblock, (d) update the FCB entry of the file, (e) return
the FCB entry index of the file.
If the entry is not found, (a) find free space to allocate the file
content (b) create FCB entry for the file, (c) return the FCB entry
index of the file
```

- fs_read

```
void fs_read(Filesystem *fs, uchar *output, int size, u32 fp);
```

The operation reads the file content into the output buffer, which follows the following logic:

```
1. find the starting address of the file content in the storage based
on the FCB entry of the file
2. read the file content into the output buffer byte by byte
```

- fs_write

```
u32 fs_write(Filesystem *fs, uchar *intput, int size, u32 fp);
// return value: execution status (0->success)
```

The operation writes the content of the input buffer to the storage, which follows the following logic:

```
1. find the starting address of the file content in the storage based
on the FCB entry of the file
2. write the content of the input buffer to the storage byte by byte
```

- RM

```
void fs_gsys(FileSystem *fs, int op, char *s);
// if op is 2, do the remove operation
```

The operation remove the file out of the file system, which follows the following logic:

```
1. find the FCB entry index of the file based on the file name
2. find the starting address of the file content in the storage and the
size of the file content base on the FCB entry
3. clear the file content in the storage, perform compaction (bit map
will be updated in compaction), update the FCB entry of the file
```

- LS

```
void fs_gsys(FileSystem *fs, int op);
// if op is 0, do LS_D (list by time), if op is 1, do LS_S (list by
size)
```

The operation will list the files in the current directory (root directory in the source part), which follows the following logic:

```
1. find all the FCB entry indexes that points to a valid file
2. list the file name in the terminal according to the information in
the FCB entry of the files
```

Note: I use bubble sort to implement the LS

Design: Bonus Part

1. Filesystem structure: FCB Entries, Superblock and Storage

In the Bonus part, the structure of superblock and storage is exactly the same as that of the source part. However, the FCB entries are modified to fit the directory structure of the file system.

- **FCB entry in Bonus part**

I expand the size of file entry to 38 bytes to store the directory information for each file. The new FCB entry structure is shown below:

```
Original file entry part: byte[31:0];
// this part is the same as the FCB entry of the source part
Directory part: byte[37:32]; // this part stores the directory information
-> directory information: byte[34:32]
    dir1: directoryIndex of depth 1 -> byte[32]
    dir2: directoryIndex of depth 2 -> byte[33]
    dir3: directoryIndex of depth 3 -> byte[34]
-> dirMap (directory index mapping): byte[35]
    // if the FCB entry corresponds to a directory, then the FCB stores a unique
directory map to indicates the directory that directory entry points at.
-> depth (directory depth): byte[36]
    // this byte records the depth of the current directory
-> dirBit (directory valid bit): byte[37]
    // this byte stores a bit to indicate that whether the FCB entry is a
directory entry (0: file entry, 1: directory entry)
```

- **Directory information of files (and directories)**

The FCB entry stores the current directory that the file is in, to get the current directory of the file, we need to know the (a) depth, (b) directory indexes of file.

Eg1: if a file has the following information in its FCB entry:

```
dir1 = 2; dir2 = 1; dir3 = 0; depth = 2; dirBit = 0;
name = "a.txt";
dirtyBit = 1;
// dir1 = 2 maps to directory with name "temp/"
// dir2 = 1 maps to directory with name "local/"
```

Then the path of file is `temp/local/a.txt`

Eg2: if a directory has the following information in its FCB entry:

```
dir1 = 2; dir2 = 1; dir3 = 0; depth = 2; dirBit = 1;
name = "low";
dirtyBit = 1;
// dir1 = 2 maps to directory with name "temp/"
// dir2 = 1 maps to directory with name "local/"
```

For a directory entry, the directory it points at is: directory it lies in + dirMap (directory index of the next directory depth).

Then the directory that the directory lies in is `temp/local`, the directory that the directory entry is pointing to is `temp/local/low`, the path of the directory is `temp/local/low`

- **Current directory pointer**

In my design, I also create a pointer named `fs->DIRECTORY` pointing to the current directory. If the user creates a file, then the information specified by the `fs->DIRECTORY` will be stored in the FCB entry of the file.

- **More remarks regarding the directory entry**

The directory corresponds to a FCB entry like a file. However, the directory does not store anything in the storage in my design. The directory only stores the (a) the directory information what the directory entry lies in, (b) the directory information that the directory entry points to.

To search for a file in a directory, we need to check (1) the file name match, (2) the directory information match.

My design of the directory structure is different from the design specified in the handout. Compared with the design in the handout (the directory stores the name of the file under the directory in the storage), my design:

```
1. may be more inefficient regarding file operations: for accessing a file under
a directory needs searching the whole FCB entry block.
3. save storage space: the directory stores nothing in the storage
4. requires more space for FCB entries: each FCB entry needs to store the
directory information of the file
```

2. File Operations

- fs_open / fs_read / fs_write / LS_S / LS_D

> These five operations are almost the same as that of the source part. The only difference is that we should perform the file operations in the current directory. We need to implement a function to check if the file is in the current directory.
>
> ```
> __device__ bool FCB_CheckDirectoryMatch_Current(FileSystem* fs, int
> index);
> ```
>
> For directory FCB entry does not store anything in storage, we do not need to care about directory entries when performing file operations.

- MKDIR

```
fs_gsys(fs, MKDIR, “app\0”);
```

This function creates a new directory, which follows the following logic:

```
1. get the current directory information
2. create a directory entry
    > find a unique dirMap for the new directory
    > create FCB entry for the directory
```

- CD

```
fs_gsys(fs, CD, “app\0”);
```

This function will move the current directory pointer to the desired directory, which follows the logic:

```
1. search through the FCB entries, find the FCB entry corresponds to
the directory name
2. get the directory information & dirMap of the FCB entry -> get the
new directory to move to
3. change the value of current directory pointer (fs->DIRECTORY) to the
new directory value
```

- CD_P

```
fs_gsys(fs, CD_P);
```

This function will move to the current directory pointer to the parent directory, which follows the logic:

```
1. get the value of current directory pointer (directory indexes,
depth)
2. move to parent directory (reset the directory index of the with the
highest depth, decrease the directory depth)
```

- PWD

```
fs, gsys(fs, PWD);
```

This function will print the path of the current directory, which follows the following logic:

```
1. store the value of current directory pointer (*(fs->DIRECTORY)) into
a local variable, create a local string stack to store directory names.
2. iterate the FCB entries, find the directory entry pointing to the
current directory, store directory name in the string stack.
3. move the current directory pointer to the parent, check if current
directory is root, if no, go to the step2.
```

- RM_RF

```
fs_gsys(fs, RM_RF, “app\0”);
```

This function remove a directory along with anything under that directory, which follows the following logic.

```
// This function is implemented with recursion
RM_RF will call a recusive function named RmDirectory(directory)
In the RmDirectory:
1. check if it is max depth (3)
2. if not max depth
    get directory information of the directory to be removed
    the iterates the FCB entries
    > if a valid file entry matches the directory information
        remove the file and do compaction
    > if a valid directory matches the directory information
        call RmDirectory(ThisDirectory) // recursion
```

Problems & Solutions

Coding Problems and Solutions

1. Program the Compaction

There can be multiple ways to implement the compaction: (option 1) compact every time a file is overwritten or removed, (option 2) compact when the storage is full.

Though the second option can be more efficient in terms of performance, it is extremely hard to test the compaction. In this case, I implement the option 1 to make debugging and testing simpler.

2. Arrange Code Structure

The code can be long (especially for the bonus task). Coding can be more efficient if I arrange the code. I arrange the code by (a) create subroutines, (b) name subroutines with indicators.

(a) Subroutines: I create subroutines for some frequently performed operations (eg: fetch dirtyBit of the FCB entry). Below are some prototypes of the subroutines.

```
__device__ u32 FCB_FindFileIndex(FileSystem* fs, char* fileName);
__device__ u32 FCB_FindFileIndex_fpointer(FileSystem* fs, u32 fpointer);
...
__device__ u32 STORAGE_FindFreeSpaceIndex(FileSystem* fs);
...
__device__ void FCB_UpdateFCBEntry_Name(FileSystem* fs, u32 FCB_index, char*
filename);
__device__ void FCB_UpdateFCBEntry_Fpointer(FileSystem* fs, u32 FCB_index, u32
fpointer);
...
__device__ char* FCB_FetchEntry_Filename(FileSystem* fs, u32 FCB_index);
__device__ u32 FCB_FetchEntry_Fpointer(FileSystem* fs, u32 FCB_index);
...
__device__ void DIR_SetDirectoryInfo(FileSystem* fs, uchar depth, uchar
directory1, uchar directory2, uchar directory3);
__device__ uchar DIR_FetchDirectory(FileSystem* fs, int depth);
```

(b) Subroutine indicators: As is shown in the above figure, the subroutines are names with indicators such as "FCB_". Doing so makes coding a little more clear.

Debugging Issues

1. Steps to Debug

My way to debug is kind of silly. I just use `printf()` everywhere & execute test program line by line with the help of `//`.

2. Memory Overflow Problem

When I test the bonus task, I found that the function exit halfway without throwing any error. The problem turns out to be there is not enough stack space to execute the program (the issue is met when I create a local u32[1024] array when sorting). To solve the program, we need to change some setting regarding the CUDA. I write the following line before the kernel execution to change the stack size for the program.

```
cudaDeviceSetLimit(cudaLimitStackSize, 32678);
// expand the stack limit to 32678 bytes, which is large enough
```

Environment & Execution

Environment

I use VS2019 and implement the assignment on my own computer. The CUDA version is 11.5, and the OS of my computer is Windows 10.

Execution and Output

Execution

To execute my program, you can use visual studio to test (press down ctrl+f5), or you can use the following approaches:

1. Type "make" (if you test in the environment of the computers of TC301)

```
(base) [cuhksz@TC-301-09 Program]$ make
nvcc -rdc=true -o main file_system.cu user_program.cu main.cu
...
./main
...
```

2. Type command (if you test in windows environment)

```
PS C:\Users\CSC3150_Assignment4\Program>nvcc -rdc=true -o main
file_system.cu main.cu user_program.cu
...
```

Output

I show the output on the Windows powershell

1. Source part

   Test case 1:

```
PS C:\Users\CSC3150_Assignment4_vs\Program> nvcc -rdc=true -o main
file_system.cu main.cu user_program.cu
file_system.cu
main.cu
user_program.cu

  正在创建库 main.lib 和对象 main.exp
PS C:\Users\CSC3150_Assignment4_vs\Program> ./main
=== 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
```

   Test case 2:

```
PS C:\Users\CSC3150_Assignment4_vs\program> ./main
=== 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
```

Note: the output of test case 3 is too long, so I will not include that in this report.

2. Bonus part

   Test case:

```
PS C:\Users\CSC3150_Assignment4_vs\Bonus> ./main
=== Sort by modified time ===
t.txt
b.txt
=== Sort by file size ===
t.txt 32
b.txt 32
=== Sort by modified time ===
app d
t.txt
b.txt
=== Sort by file size ===
t.txt 32
b.txt 32
app 0 d
=== Sort by file size ===
=== Sort by file size ===
a.txt 64
b.txt 32
soft 0 d
=== Sort by modified time ===
soft d
```

```
b.txt
a.txt
/app/soft
=== Sort by file size ===
B.txt 1024
C.txt 1024
D.txt 1024
A.txt 64
=== Sort by file size ===
a.txt 64
b.txt 32
soft 0 d
/app
=== Sort by file size ===
t.txt 32
b.txt 32
app 6 d
=== Sort by file size ===
a.txt 64
b.txt 32
soft 0 d
=== Sort by file size ===
t.txt 32
b.txt 32
app 6 d
```

Some Thoughts

In assignment 4, I:

1. Implement a simple file system, have better understanding about the file system and directory structure in the operating system.
2. Try to implement a different approach of directory structure from the approach in hand-out, have insights in how different implementation affects the performance and efficiency.
3. Write program with many subroutines, learn the importance of encapsulating.
4. Debug the program for quite long time, get more accustomed to debugging.
5. Encounter the CUDA stack overflow problem, learn something regarding CUDA memory management.

I find that assignment 4 is not hard once I figure out the logic. To understand the logic of the program is the first step to write a code. Everything can be implemented if we keep the logic organized in our brain.

Thanks for being patient to grade my programs and report! Have a nice day!