

CSC3150 Project4 Report

118010045 Cui Yuncong

November 2020

Contents

1	Introduction	1
2	Design	2
2.1	Program Flow	2
2.2	Functions	4
3	Problem and Solution	8
3.1	Segmentation Management	8
3.2	Sort	8
3.3	Storage Problem	9
4	Running Environment	9
5	Execution and Program Output	9
5.1	Test Case 1	10
5.2	Test Case 2	10
5.3	Test Case 3	11
6	Learning Remarks	11
6.1	Space Management	11
6.2	Recursion in CUDA	11
6.3	File System	12

1 Introduction

File systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file with its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system on to the physical secondary-storage devices.

The file-organization module knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block address to physical block address for the basic file system to transfer. Each file's logical blocks are numbered from 0 (or 1) through N . Since the physical blocks containing the data usually do not match the logical numbers, a translation is required to locate each block. The logical file system manages metadata information. Metadata includes all of the file-system structure except the actual data (or contents of the files). The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file- organization module when requested.

The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file-control blocks. A file-control block (FCB) (an inode in UNIX file systems) contains information about the file, including ownership, permissions, and location of the

file contents. The program is implemented on CUDA and tested on the windows OS with CUDA 11.1, VS 2019, and NVIDIA GeForce GTX 1060.

2 Design

2.1 Program Flow

The program consists of 6 parts:

1. The structure for volume in the program is designed as following. The super block part contains 4KB, each bit represents to one block, if the block contains data, the bit will be 1, otherwise the bit will be 0. The FCB file will contain the file information including size, name, create date, modified date and the block position for the start point. For each file the size information stores in the 1-4 bytes of the FCB file, the name stores in the 5-24, create date stores in the 25-26 bytes, the modified date stores in the 27-28 blocks and the block position stores in the 29-32 bytes. Each file in total occupies 32 bytes in the FCB parts. The last 1024KB is used to store the data information.

2. Before the read or write operation, firstly call the open operation. In the open operation, the file will be checked if it exists or not. If the file doesn't exist, new space in the FCB will be found to store the name, create date, modified date, start block position in it, find a new pace in the storage and return the block position. If the file existed, we will find its FCB position and in the FCB position we will find its block position for the start point.

Then, if the user requires to write data in the existing file, the program will cleans the original file data in the storage and change the bit for corresponding blocks in to 0 in the super block.

3. In the read operation, the program reads the file from the storage and writes the data in the output file.

4. In the write operation, the program will first check if there are enough space to write the file. If the space is not enough, the program will assign a new position for the file. If the data exceed the storage space the program will return the error signal. Then the system will load the data in the valid position in the storage and corresponding load or change the size information in the FCB.

5. The program provides two sort methods, the LS_D operation and the LS_S operation. In the LS_D operation, the file will be sorted and displayed by the modified time. The recent modified file will be displayed first. In the LS_S operation, the file will be sorted by the size. The larger size file will be displayed at first. For the files with same size, the file with earlier create time will be displayed at first.

6. The program also provides the RM operation to remove the file from the file system. The FCB position and the block position are corresponding to file. The file data in the storage and the information stored in the FCB will be cleaned. Also, the bit for the file will be changed to 0 in the super block.

After the write, if the external segmentation occurred, the program will do the segmentation management automatically to make sure that there is no external segment in the storage. The program will also delete the external segment in the FCB.

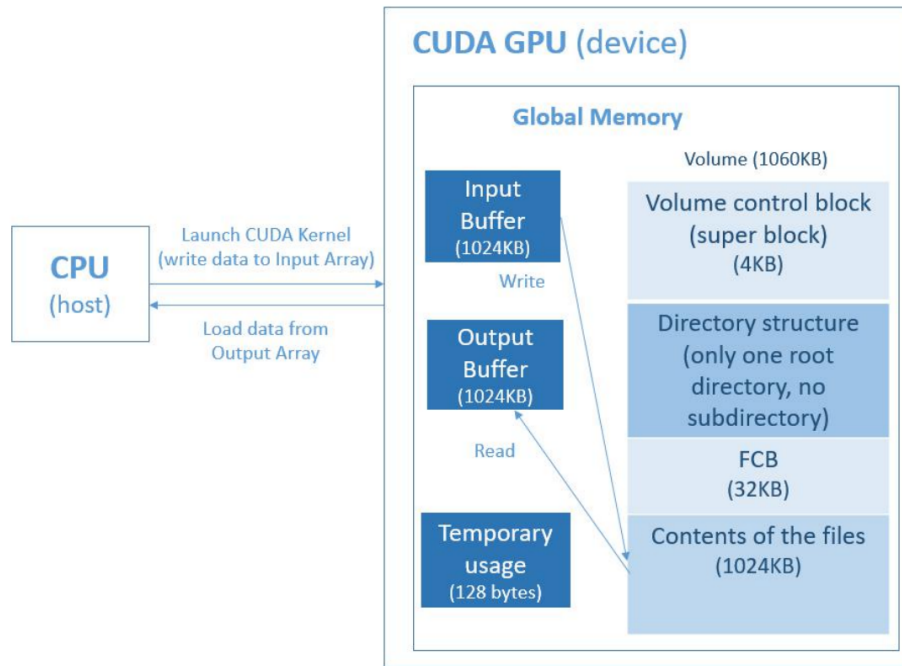


Figure 1: Memory Structure

2.2 Functions

This section describes the functions in the file_system.cu.

1. fs_open()

The fs_open() function can assign a starting block for the file and store the basic information except size when the user call the open operation.

```

__device__ u32 fs_open(FileSystem *fs, char *s, int op)
{
    int i;
    if (if_exist(fs, s) == -1)
    {
        if (!op)
        {
            printf("File Open Failed\n");
            return -1;
        }
        current_FCB_position = FCB_position;
        for (i = 4; i < 24; i++)
        {
            fs->volume[FCB_position + i] = s[i - 4];
            fs->volume[FCB_position + i] = gtime_create >> 8;
            fs->volume[FCB_position + i + 1] = gtime_create;
            fs->volume[FCB_position + i + 2] = gtime >> 8;
            fs->volume[FCB_position + i + 3] = gtime;
            fs->volume[FCB_position + i + 4] = block_position >> 24;
            fs->volume[FCB_position + i + 5] = block_position >> 16;
            fs->volume[FCB_position + i + 6] = block_position >> 8;
            fs->volume[FCB_position + i + 7] = block_position;
            gtime_create++;
            gtime++;
            FCB_position = FCB_position + 32;
            return block_position;
        }
    }
    else
    {
        current_FCB_position = if_exist(fs, s);
        u32 start_block = (fs->volume[current_FCB_position + 28] << 24) + (fs->volume[current_FCB_position + 29] << 16)
            + (fs->volume[current_FCB_position + 30] << 8) + (fs->volume[current_FCB_position + 31]);
        if (op == 1)
        {
            u32 size = (fs->volume[current_FCB_position] << 24) + (fs->volume[current_FCB_position + 1] << 16)
                + (fs->volume[current_FCB_position + 2] << 8) + (fs->volume[current_FCB_position + 3]);
            for (i = 0; i < size; i++)
            {
                fs->volume[start_block * 32 + i + fs->FILE_BASE_ADDRESS] = 0;
            }
            for (i = 0; i < (size - 1) / 32 + 1; i++)
            {
                u32 super_block_position = start_block + i;
                int shift_number = super_block_position % 8;
                fs->volume[super_block_position / 8] = fs->volume[super_block_position / 8] - (1 << shift_number);
            }
            fs->volume[current_FCB_position + 26] = gtime >> 8;
            fs->volume[current_FCB_position + 27] = gtime;
            gtime++;
        }
        return start_block;
    }
}

```

Figure 2: fs_open() Function

2. fs_read()

The fs_read() function is to implement the read operation while the user called. It will continuously put the data from the pointer position from the storage into the output buffer.

```

__device__ void fs_read(FileSystem *fs, uchar *output, u32 size, u32 fp)
{
    for (int i = 0; i < size; i++)
        output[i] = fs->volume[fp * 32 + i + fs->FILE_BASE_ADDRESS];
}

```

Figure 3: fs_read() Function

3. fs_write()

The `fs_write()` function is used to implement the write operation while the user called. It will write the data from the input buffer in to the storage in the valid block.

```

_device__ u32 fs_write(FileSystem *fs, uchar* input, u32 size, u32 fp)
{
    int i;
    if ((fs->volume[(fp + (size - 1) / 32) / 8] >> (fp + (size - 1) / 32) % 8) % 2 == 0)
    {
        u32 old_file_size = (fs->volume[current_FCB_position] << 24) + (fs->volume[current_FCB_position + 1] << 16) + (fs->volume[current_FCB_position + 2] << 8) + (fs->volume[current_FCB_position + 3]);
        u32 original_size = old_file_size - size;
        for (i = 0; i < size; i++)
        {
            fs->volume[fp * 32 + i + fs->FILE_BASE_ADDRESS] = input[i];
            if (i % 32 == 0)
            {
                u32 super_block_position = fp + i / 32;
                int shift_number = super_block_position % 8;
                fs->volume[(fp + i / 32) / 8] = fs->volume[(fp + i / 32) / 8] + (1 << shift_number);
            }
        }
        if (int(original_size) < 0)
            block_position = block_position + (-original_size - 1) / 32 + 1;
        for (i = 0; i < 4; i++)
            fs->volume[current_FCB_position + i] = size >> (3 - i) * 8;
        if (original_size > 0 && old_file_size != 0 && fp != block_position - 1)
            segment_management(fs, fp + (size - 1) / 32 + 1, original_size);
    }
    else
    {
        u32 original_size = (fs->volume[current_FCB_position] << 24) + (fs->volume[current_FCB_position + 1] << 16) + (fs->volume[current_FCB_position + 2] << 8) + (fs->volume[current_FCB_position + 3]);
        if (block_position * 32 - 1 + size >= fs->SUPERBLOCK_SIZE)
            return -1;
        else
        {
            for (i = 0; i < size; i++)
            {
                fs->volume[block_position * 32 + i + fs->FILE_BASE_ADDRESS] = input[i];
                if (i % 32 == 0)
                {
                    u32 super_block_position = block_position + i / 32;
                    int shift_number = super_block_position % 8;
                    fs->volume[(block_position + i / 32) / 8] = fs->volume[(block_position + i / 32) / 8] + (1 << shift_number);
                }
            }
            for (i = 0; i < 4; i++)
                fs->volume[current_FCB_position + i] = size >> (3 - i) * 8;
            fs->volume[current_FCB_position + 28] = block_position >> 16;
            fs->volume[current_FCB_position + 29] = block_position >> 16;
            fs->volume[current_FCB_position + 30] = block_position >> 8;
            fs->volume[current_FCB_position + 31] = block_position;
            segment_management(fs, fp, original_size);
        }
    }
}

```

Figure 4: `fs_write()` Function

4. `fs_gsys(FileSystem *fs, uchar *output, u32 size, u32 fp)`

The `fs_gsys()` function is used to implement the LS_S and LS_D operation. It will firstly find the index range for the valid file system information and call the `filesort()` function to sort the FCB according to size or date.


```

__device__ void fs_gsys(FileSystem *fs, int op, char *s)
{
    if (if_exist(fs, s) == -1)
        return;
    current_FCB_position = if_exist(fs, s);
    u32 start_block = (fs->volume[current_FCB_position + 28] << 24) + (fs->volume[current_FCB_position + 29] << 16) + (fs->volume[current_FCB_position + 30] << 8) + (fs->volume[current_FCB_position + 31]);
    u32 size = (fs->volume[current_FCB_position] << 24) + (fs->volume[current_FCB_position + 1] << 16) + (fs->volume[current_FCB_position + 2] << 8) + (fs->volume[current_FCB_position + 3]);
    for (int i = 0; i < size; i++)
        fs->volume[start_block * 32 + i + fs->FILE_BASE_ADDRESS] = 0;
    for (int i = 0; i < (size - 1) / 32 + 1; i++)
        fs->volume[start_block + i] = 0;
    for (int i = 0; i < 32; i++)
        fs->volume[current_FCB_position + i] = 0;
    segment_management(fs, start_block, size);
    for (int i = current_FCB_position; i < 36863; i = i + 32)
    {
        if (fs->volume[i + 32] == 0 && fs->volume[i + 32 + 1] == 0 && fs->volume[i + 32 + 2] == 0 && fs->volume[i + 32 + 3] == 0)
            break;
        for (int j = 0; j < 32; j++)
        {
            fs->volume[i + j] = fs->volume[i + j + 32];
            fs->volume[i + j + 32] = 0;
        }
    }
    FCB_position = FCB_position - 32;
}

```

Figure 5: fs_gsys(FileSystem *fs, uchar *output, u32 size, u32 fp) Function

5. fs_gsys(FileSystem *fs, int op)

The fs_gsys() function is used to implement the remove operation while called. It will clean the old file data in the storage, clean the block in the super block, clean the file information in the FCB.

```

__device__ void fs_gsys(FileSystem *fs, int op)
{
    u32 end;
    for (u32 i = 4096; i < 36895; i = i + 32)
    {
        u32 size = (fs->volume[i] << 24) + (fs->volume[i + 1] << 16) + (fs->volume[i + 2] << 8) + (fs->volume[i + 3]);
        end = i - 32;
        if (!size)
        {
            size = (fs->volume[4096] << 24) + (fs->volume[4096 + 1] << 16) + (fs->volume[4096 + 2] << 8) + (fs->volume[4096 + 3]);
            break;
        }
    }
    if (end < 4096)
        return;
    filesort(fs, 4096, end, op);
    display(fs, end, op);
}

```

Figure 6: fs_gsys(FileSystem *fs, int op)Function

6. filesort()

The filesort() function is used to selection_sort the original FCB by modified time if the operation is LS_D or by file size if the operation is LS_S.

7. check():

The `check()` function is used to find the corresponding FCB position by name. The function will return the FCB position if the file is found and return -1 if the file is not found.

8. `segment_management()`

The `segment_management()` function is used to delete the external segment in storage. It will also automatically change the information in FCB for any file stored after the external segment.

3 Problem and Solution

3.1 Segmentation Management

The file in the existing file and the new file size exceed the old file size, the block needs to be reallocated for the new file. This will cause the external segment problem in the storage. Therefore, the programs uses the segmentation management every time to reuse the external segmentation.

3.2 Sort

Quicksort can reduce the execution time. However, the program cannot hold down because the CUDA does not support recursion for too many times. Then the program changes the `quick_sort` into `bubble_sort`. However, the `bubble_sort` is based on the exchange. For sorting the file in this project, it is time-consuming. Therefore, the `bubble_sort` has been changed to the selection-sort, which is a sort based on the comparison.

3.3 Storage Problem

Most of the information in the FCB file need cannot be stored in only 1 byte. 2 bytes or more to store the data are needed. Thus, the data will be divided into parts with each part has one byte.

4 Running Environment

Version of OS: Window10

IDE: Visual Studio 2019

Video Card: NVIDIA Geforce GTX 1060

CUDA Cersion: 11.1

```
C:\WINDOWS\system32\cmd.exe
C:\Program Files\NVIDIA Corporation\NVSMI>nvidia-smi
Tue Oct 27 21:38:24 2020
```

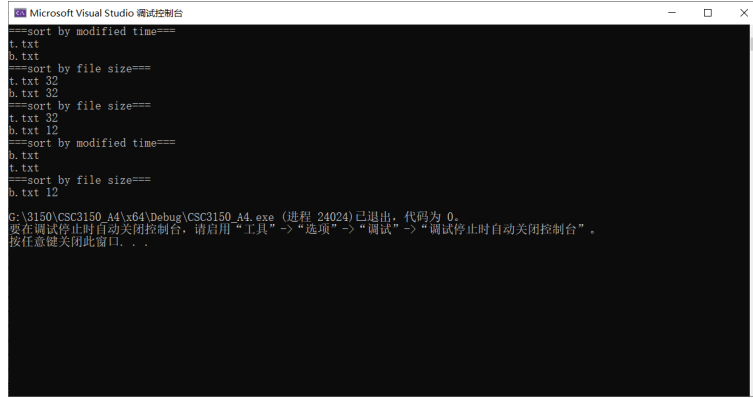
NVIDIA-SMI 456.43				Driver Version: 456.43				CUDA Version: 11.1			
GPU	Name	TCC/WDDM		Bus-Id	Disp. A	Volatile	Uncorr. ECC				
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.				
0	GeForce GTX 1060	WDDM		00000000:01:00.0	Off		N/A				
N/A	68C	P2	35W / N/A	212MiB / 6144MiB		100%	Default				

Figure 7: GPU Information and CUBA Version

5 Execution and Program Output

Use Visual Stdio to open the project. Press ‘Ctrl’ + ‘F7’ to compile each CUDA file and use the ‘Ctrl’ + ‘F5’ to execute the program.

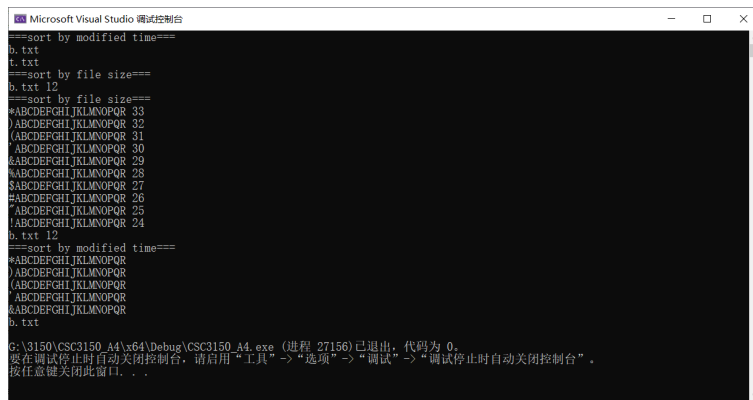
5.1 Test Case 1



```
Microsoft Visual Studio 调试控制台
===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
G:\3150\CSC3150_A4\x64\Debug\CSC3150_A4.exe (进程 24024) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

Figure 8: Compile Successfully

5.2 Test Case 2



```
Microsoft Visual Studio 调试控制台
===sort by modified time===
b.txt
t.txt
===sort by file size===
b.txt 12
===sort by file size===
*ABCDEFGHIJKLMNQPQR 33
)ABCDEFGHIJKLMNQPQR 32
(ABCDEFGHIJKLMNQPQR 31
'ABCDEFGHIJKLMNQPQR 30
&ABCDEFGHIJKLMNQPQR 29
*ABCDEFGHIJKLMNQPQR 28
&ABCDEFGHIJKLMNQPQR 27
*ABCDEFGHIJKLMNQPQR 26
'ABCDEFGHIJKLMNQPQR 25
)ABCDEFGHIJKLMNQPQR 24
b.txt 12
===sort by modified time===
*ABCDEFGHIJKLMNQPQR
)ABCDEFGHIJKLMNQPQR
(ABCDEFGHIJKLMNQPQR
'ABCDEFGHIJKLMNQPQR
&ABCDEFGHIJKLMNQPQR
b.txt
G:\3150\CSC3150_A4\x64\Debug\CSC3150_A4.exe (进程 27156) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

Figure 9: Page Fault Number

5.3 Test Case 3

Figure 10: Snapshot

6 Learning Remarks

6.1 Space Management

To manage the external segment, use the segment management when the storage is out of available space. Also, doing the segment management when the external segment occurred still works.

6.2 Recursion in CUDA

When the program tries to use quick_sort by recursion, once the stack is out, the results could be cleared. This is because when the recursion occurred , the data will be stored in stack which may cause conflict in the CUDA.

6.3 File System

A bitmap or bitvector in the super block to record the used block can support the contiguous allocation work in the file system. To be specific, a new block at the end of the used block sequences is needed if the original block size is not enough for new file.