# Report

**Li Huayue 118010138**

## 1 The thought of design:

After reading the requirements of the assignment, my whole thought is to finish the functions which are shown as follows:

```
__device__ u32 fs_open(FileSystem *fs, char *s, int op);
__device__ void fs_read(FileSystem *fs, uchar *output, u32 size, u32 fp);
__device__ u32 fs_write(FileSystem *fs, uchar* input, u32 size, u32 fp);
__device__ void fs_gsys(FileSystem *fs, int op);
__device__ void fs_gsys(FileSystem *fs, int op, char *s);
```

What is more is that I also create a lot of functions of my own to help to achieve these functions. Now I will introduce every part of my project following the requirements of this assignment.

I have achieved the file volume structure using the template of this assignment. In my project, the size of my file volume structure is 1085440 bytes:

```
#define VOLUME_SIZE 1085440 //4096+32768+1048576
```

We can use a pointer $fs \rightarrow volume$ to get all the contents in the file volume structure. And in this file volume structure, according to the hints in the assignment introduction, I divided it into three parts:

(1) Because the last 1024KB of the file volume structure are divided into blocks where every block is 32 bytes, we need some space used to record which blocks are used and which blocks are not used. Therefore, the first part of the file volume is used to achieve this function which is called “super block part”.

(2) Because the system we write is a file system. To manage all the files in the file

system better, we need to record all the information for every file in the file system like the name of the file, the address of the file content stored in the file system, the size of the file, the create time of the file and the modified time of the file. Therefore, the second part of the file volume is used to achieve this function which is called "FCB part".

(3) Because the core of file system is to store contents of all the files in it, we need some space to store all the contents of files in the file system. Therefore, the third part of the file volume is used to achieve this function which is called "storage part".

That is the whole thought of mine to implement the file volume structure.

Now I will discuss how to implement the free space management. Because the time complexity of bit-vector or bit-map is too large and the space to achieve we can use is a little small (which is only 4KB), the implement of bit-vector and bit-map are complex. After observation, I found another easier way. I have found that in this assignment, the way of store we use is contiguous allocation and we store the contents of files block by block. Therefore, we can just use a number to implement the free space management. When we add contents into the file system, we will add them at the end of the storage and when we delete contents in the file system, the number of blocks decreased is fixed (which is equal to $(size/32 + 1)$) no matter where the file address is. Therefore, using only one number (I use 4 bytes in the file system to store it) in the file volume structure to implement free space management is enough. I have stored the special number into the file volume structure and the program can get it using the "$get\_block\_used\_number$" function:

```
__device__ u32 get_block_used_number(FileSystem *fs) {
        u32 number;
        if (fs->volume[0] == NULL) {
                return 0;
        }
        for (u32 i = 0; i < 4; i++) {
                number += (fs->volume[i] << (i * 8));
        }
        return number;
}
```

The program can also update the special number using the "*update_super_block*" function:

```
__device__ void update_super_block(FileSystem *fs, u32 number) {
        int number_RSB_1 = number - ((number >> 8) << 8);
        int number_RSB_2 = ((number - ((number >> 16) << 16) - number_RSB_1) >> 8);
        int number_RSB_3 = ((number - ((number >> 24) << 24) - number_RSB_1 - (number_RSB_2 << 8)) >> 16);
        int number_RSB_4 = number >> 24;
        fs->volume[0] = number_RSB_1;
        fs->volume[1] = number_RSB_2;
        fs->volume[2] = number_RSB_3;
        fs->volume[3] = number_RSB_4;
}
```

From the special number we can know which blocks are used and which blocks are not used in the contiguous allocation. That is the whole thought of mine to implement the contiguous allocation.

Now I will discuss how to implement the contiguous allocation. Because the "storage part" has been divided into blocks. When a new file is written into the file system, the content of it must begin to store at the beginning of one of the blocks which means that there is no case that the contents of different files happen in the same block. And we can use the special number in the "super block part" to get the address of corresponding block address to write new contents into the file system. Therefore, the contiguous allocation means "block contiguous" but not "byte contiguous". For example:

```
u32 new_addr = fs->FILE_BASE_ADDRESS + used_block_number * 32;
for (u32 i = 0; i < size; i++) {
        fs->volume[new_addr + i] = input[i];
}
```

And when some contents in the file system are deleted, all the contents after those contents in the file system will compact to make sure that the rest of the contents in the file system is contiguous allocation that is also why I mean that using only one number to implement free space management is enough. For example:

```
u32 old_file_address_2 = old_file_address + decrease_block_number * 32;
u32 compact_time = 1085440 - old_file_address_2;
for (u32 i = 0; i < compact_time; i++) {
        fs->volume[old_file_address + i] = fs->volume[old_file_address_2 + i];
        fs->volume[old_file_address_2 + i] = NULL;
}
```

In this way, we can achieve the continuous storage for the contents in the file system and update the free space management number together. That is the whole thought of mine to implement the contiguous allocation.

Now I will discuss how to implement the "$fs\_open\ operation$". After observation, I have found that there are two kinds of open operations: read and write. Therefore, I create a "if-else" structure to judge and make the program to execute corresponding functions:

```
if (op == G_READ) {

else if (op == G_WRITE)
```

The core thought of my "$fs\_open$" function is to return a $fp$ (which is the address of the corresponding $fcb$ address of corresponding file). I also implement a function of my own called "$find\_fp\_using\_name$" to find the $fp$ using the file name which is one of the input parameters.

```
__device__ u32 find_fp_using_name(FileSystem *fs, char *s) {
	for (u32 i = 0; i < fs->FCB_ENTRIES; i++) {
		u32 fcb_block_start = fs->SUPERBLOCK_SIZE + i * fs->FCB_SIZE;
		char file_name[20];
		for (u32 j = 0; j < 20; j++) {
			char part = fs->volume[fcb_block_start + j];
			file_name[j] = part;
		}
		u32 length = 0;
		while (true) {
			if (s[length] != '\0') {
				length += 1;
			}
			else {
				break;
			}
		}
		bool condition = true;
		for (u32 j = 0; j < length; j++) {
			if (file_name[j] != s[j]) {
				condition = false;
			}
		}
		if (condition == false) {
			continue;
		}
		else if (condition == true)
		{
			return fcb_block_start;
		}
	}
	return -1;
}
```

In this function, the program will compare the input file name with all the file names in the file system byte by byte and then return the corresponding $fp$ address if one of them is matched.

If the program wants to open the file in a "read" way, the program will first use the "$find\_fp\_using\_name$" function to get the $fp$ of that file and then return it, if they do not find the file, the program will return $-1$.

```
if (op == G_READ) {
		//printf("this is the open of read\n");
		u32 fp = find_fp_using_name(fs, s);
		if (fp != -1) {
			//printf("we have find the file and the fp is:%d\n", fp);
			return fp;
		}
		else {
			//printf("there is no such file!\n");
			return -1;
		}
}
```

If the program wants to open the file in a "write" way, the program will judge whether the file has existed in the file system or not. If the file has already existed, the program will find the $fp$ of the file and then return it. If the file does not exist in the file system, the program will find free $fcb$ space for the program and then update the $fcb$ space and then return the new $fp$ for the file.

```
else if (op == G_WRITE)
{
	//printf("this is the open of write\n");
	u32 fp = find_fp_using_name(fs, s);
	if (fp != -1) {
		//printf("we have find the file and the fp is:%d\n", fp);
		return fp;
	}
	else {
		//printf("there is no such file but we will create a file in the fcb!\n");
		u32 fcb_block_start = 0;
		for (u32 i = 0; i < fs->FCB_ENTRIES; i++) {
			fcb_block_start = fs->SUPERBLOCK_SIZE + i * fs->FCB_SIZE;
			if (fs->volume[fcb_block_start] != NULL) {
				continue;
			}
			else {
				created_time += 1;
				u32 used_block_number = get_block_used_number(fs);
				//printf("the created time has updated and now it is:%d\n", created_time);
				u32 new_addr = fs->FILE_BASE_ADDRESS + used_block_number * 32;
				u32 length = 0;
				while (true) {
					if (s[length] != '\0') {
						length += 1;
					}
					else {
						break;
					}
				}
				for (u32 i = 0; i < length; i++) {
					fs->volume[fcb_block_start + i] = s[i];
				}
				update_fcb(fs, fcb_block_start, new_addr, 0, 1);
				break;
			}
		}
		//printf("we have found an empty fcb and the address is:%d\n", fcb_block_start);
		return fcb_block_start;
	}
```

That is the whole thought of mine to implement the "$fs\_open\ operation$". What you should pay attention is that because the information of every file includes the create time which will be used in the sort, the only place where the create time will update is the place that when we open a file with write way and the file does not exist in the file system.

Now I will discuss how to implement the "$fs\_write\ operation$". After getting the $fp$, if we want to write a file in the file system, there are two kinds of cases:

(1) If the file does not exist in the file system, we will write the new contents into the "storage" part" of the file system and update the "$fcb\ part$" and "super block part" according to the input parameters:

```
u32 file_address = get_address_from_FCB(fs, fp);
modified_time += 1;
if (fs->volume[file_address] == NULL) {
        u32 increase_block_number = (size / 32 + 1);
        u32 used_block_number = get_block_used_number(fs);
        used_block_number += increase_block_number;
        update_super_block(fs, used_block_number);
        for (u32 i = 0; i < size; i++) {
                fs->volume[file_address + i] = input[i];
        }
        update_fcb(fs, fp, file_address, size, 2);
}
```

By the way, I have created a lot of functions to get information from the "$fcb\ part$" which are shown as follows:

```
__device__ u32 get_address_from_FCB(FileSystem *fs, u32 fp)
__device__ u32 get_size_from_FCB(FileSystem *fs, u32 fp)
__device__ u32 get_create_time_from_FCB(FileSystem *fs, u32 fp)
__device__ u32 get_modified_time_from_FCB(FileSystem *fs, u32 fp)
```

(2) If the file has already existed, the program will first delete all the contents in the file system, update the "$super\ block\ part$" (decrease the $old\ size/32 + 1$). And then compacting the "storage part". After finishing the whole task of deleting, the program will then write the new input contents of the file at the new address and then update the "$super\ block\ part$" (increase the $new\ size/32 + 1$) and update

the "$fcb\ part$" together:

```
else {
        u32 old_file_address = get_address_from_FCB(fs, fp);
        u32 old_file_size = get_size_from_FCB(fs, fp);
        u32 decrease_block_number = (old_file_size / 32 + 1);
        u32 used_block_number = get_block_used_number(fs);
        used_block_number -= decrease_block_number;
        update_super_block(fs, used_block_number);
        u32 old_file_address_2 = old_file_address + decrease_block_number * 32;
        u32 compact_time = 1085440 - old_file_address_2;
        for (u32 i = 0; i < compact_time; i++) {
                fs->volume[old_file_address + i] = fs->volume[old_file_address_2 + i];
                fs->volume[old_file_address_2 + i] = NULL;
        }

        u32 new_addr = fs->FILE_BASE_ADDRESS + used_block_number * 32;
        for (u32 i = 0; i < size; i++) {
                fs->volume[new_addr + i] = input[i];
        }
        u32 increase_block_number = (size / 32 + 1);
        used_block_number += increase_block_number;
        update_super_block(fs, used_block_number);
        update_fcb(fs, fp, new_addr, size, 2);
}
```

That is the whole thought of mine to implement the "$fs\_write\ operation$". What you should pay attention is that because the information of every file includes the modified time which will be used in the sort, the only place where the modified time will update is the place that when we call the write function for a file. Therefore, the modified time will update in the write function:

```
modified_time += 1;
```

Now I will discuss how to implement the "$fs\_read\ operation$". After getting the $fp$ from the open operation, if the program wants to read the file, the program will first judge whether the file exists or not. If the value of $fp$ is $-1$ which means the file does not exist, the program will show the information to the user and finish the function. If the value of $fp$ is not $-1$ which means the file exists, the program will get necessary information from the "$fcb\ part$" and then read the contents to the output

byte by byte:

```
__device__ void fs_read(FileSystem *fs, uchar *output, u32 size, u32 fp)
{
	/* Implement read operation here */
	//printf("this is the process of read\n");
	if (fp == -1) {
		printf("there is no such file so you can not read\n");
	}
	else {
		u32 file_address = get_address_from_FCB(fs, fp);
		for (u32 i = 0; i < size; i++) {
			output[i] = fs->volume[file_address + i];
		}
	}
}
```

That is the whole thought of mine to implement the "$fs\_read\ operation$".

Now I will discuss how to implement the "$fs\_gsys(RM)\ operation$". The core thought of this function is to delete a file in the file system. After getting the $fp$, the program will first judge whether the file exists in the file system or not. If the file does not exist, the program will mention the user and finish the function. If the file exists in the file system, the function will delete the contents of that file in the "storage part" and then compact the "storage part" and update the "super block" part and update the "$fcb\ part$" together.

```
__device__ void fs_gsys(FileSystem *fs, int op, char *s)
{
	/* Implement rm operation here */
	//printf("this is the process of RM\n");
	u32 fp = find_fp_using_name(fs, s);
	if (fp == -1) {
		printf("there is no such file so you cannot remove it\n");
	}
	else {
		u32 file_size = get_size_from_FCB(fs, fp);
		u32 decrease_block_number = file_size / 32 + 1;
		u32 used_block_number = get_block_used_number(fs);
		used_block_number -= decrease_block_number;
		update_super_block(fs, used_block_number);

		u32 file_address = get_address_from_FCB(fs, fp);
		u32 file_address_2 = file_address + decrease_block_number * 32;
		u32 compact_time = 1085440 - file_address_2;
		for (u32 i = 0; i < compact_time; i++) {
			fs->volume[file_address + i] = fs->volume[file_address_2 + i];
			fs->volume[file_address_2 + i] = NULL;
		}

		for (u32 i = 0; i < 32; i++) {
			fs->volume[fp + i] = NULL;
		}
	}
}
```

That is the whole thought of mine to implement the "$fs\_gsys(RM)\ operation$".

Now I will discuss how to implement the "$fs\_gsys(LS\_D)\ operation$" and "$fs\_gsys(LS\_S)\ operation$" together because they are in the same function.

The core thought of "$fs\_gsys(LS\_D)\ operation$" is to sort all the files in the file system according to their modified time and then list them in the terminal (the last modified file will be first listed). Here I use two arrays: one is used to store all the fps of every file and the other is used to store all the modified time of every file and the index of the two arrays are corresponding. Then we use the "$bubble\ sort$" to sort the modified time array. When the modified time array changes, we change the fps list together for the same index. When the sort finishes, the fps list is also sorted according to the sort of the modified time list. Finally, we will get the file names according to the fps in the fps list and then print them in the terminal.

```
if (op == LS_D) {
        printf("===sort by modified time===\n");
        u32 file_name_fp_list[1024];
        u32 index = 0;
        for (u32 i = 0; i < 1024; i++) {
                u32 fcb_block_start = fs->SUPERBLOCK_SIZE + i * fs->FCB_SIZE;
                if (fs->volume[fcb_block_start] != NULL) {
                        file_name_fp_list[index] = fcb_block_start;
                        index += 1;
                }
        }

        u32 file_modified_time_list[1024];
        u32 file_modified_time = 0;
        for (u32 i = 0; i < index; i++) {
                file_modified_time = get_modified_time_from_FCB(fs, file_name_fp_list[i]);
                file_modified_time_list[i] = file_modified_time;
        }

        for (u32 i = 0; i < index - 1; i++) {
                for (u32 j = 0; j < index -1 - i; j++) {
                        if (file_modified_time_list[j] < file_modified_time_list[j + 1]) {
                                u32 temp_1 = file_modified_time_list[j];
                                file_modified_time_list[j] = file_modified_time_list[j + 1];
                                file_modified_time_list[j + 1] = temp_1;
                                u32 temp_2 = file_name_fp_list[j];
                                file_name_fp_list[j] = file_name_fp_list[j + 1];
                                file_name_fp_list[j + 1] = temp_2;
                        }
                }
        }

        for (u32 i = 0; i < index; i++) {
                u32 fcb_address = file_name_fp_list[i];
                char file_name[20];
                for (u32 j = 0; j < 20; j++) {
                        //printf("womenjinxingdaozheyibule");
                        char part = fs->volume[fcb_address + j];
                        file_name[j] = part;
                }
                printf("%s\n", file_name);
        }
}
```

That is the whole thought of mine to implement the "$fs\_gsys(LS\_D)\ operation$". The core thought of "$fs\_gsys(LS\_S)\ operation$" is to sort all the files according to the size of them and if the size of some files are the same, the file which is created first will be listed first). Similar to the "$fs\_gsys(LS\_D)\ operation$", I create three arrays: one is the fps list, one is the size list and one is the create time list. I will first sort the size list using "bubble sort" and when the size list change, the fps list will also change for the same index which is similar to the operation in the "$fs\_gsys(LS\_D)\ operation$".

```
else if (op == LS_S) {
        printf("===sort by file size===\n");
        u32 file_name_fp_list[1024];
        u32 index = 0;
        for (u32 i = 0; i < 1024; i++) {
                u32 fcb_block_start = fs->SUPERBLOCK_SIZE + i * fs->FCB_SIZE;
                if (fs->volume[fcb_block_start] != NULL) {
                        file_name_fp_list[index] = fcb_block_start;
                        index += 1;
                }
        }

        u32 file_size_list[1024];
        u32 file_size = 0;
        for (u32 i = 0; i < index; i++) {
                file_size = get_size_from_FCB(fs, file_name_fp_list[i]);
                file_size_list[i] = file_size;
        }

        u32 file_create_time_list[1024];
        u32 file_create_time = 0;
        for (u32 i = 0; i < index; i++) {
                file_create_time = get_create_time_from_FCB(fs, file_name_fp_list[i]);
                file_create_time_list[i] = file_create_time;
        }

        for (u32 i = 0; i < index - 1; i++) {
                for (u32 j = 0; j < index - 1 - i; j++) {
                        if (file_size_list[j] < file_size_list[j + 1]) {
                                u32 temp_1 = file_size_list[j];
                                file_size_list[j] = file_size_list[j + 1];
                                file_size_list[j + 1] = temp_1;
                                u32 temp_2 = file_name_fp_list[j];
                                file_name_fp_list[j] = file_name_fp_list[j + 1];
                                file_name_fp_list[j + 1] = temp_2;
                                u32 temp_3 = file_create_time_list[j];
                                file_create_time_list[j] = file_create_time_list[j + 1];
                                file_create_time_list[j + 1] = temp_3;
                        }
                }
        }
```

Because there exists the case that some files have the same size, according to the rule

of "$fs\_gsys(LS\_S)\ operation$". I sort the create time list using the bubble sort but what we should pay attention is that because this sort is the "second sort", we only need to sort when the size of files of the neighboring fps in the fps list are the same. Finally, we will get the file names according to the fps in the fps list and then print them in the terminal. We also print the sizes of corresponding files.

```
for (u32 i = 0; i < index - 1; i++) {
        for (u32 j = 0; j < index - 1 - i; j++) {
                if ((file_create_time_list[j] > file_create_time_list[j + 1])&&(file_size_list[j] == file_size_list[j + 1])) {
                        u32 temp_1 = file_size_list[j];
                        file_size_list[j] = file_size_list[j + 1];
                        file_size_list[j + 1] = temp_1;
                        u32 temp_2 = file_name_fp_list[j];
                        file_name_fp_list[j] = file_name_fp_list[j + 1];
                        file_name_fp_list[j + 1] = temp_2;
                        u32 temp_3 = file_create_time_list[j];
                        file_create_time_list[j] = file_create_time_list[j + 1];
                        file_create_time_list[j + 1] = temp_3;
                }
        }
}
for (u32 i = 0; i < index; i++) {
        u32 fcb_address = file_name_fp_list[i];
        char file_name[20];
        for (u32 j = 0; j < 20; j++) {
                //printf("womenjinxingdaozheyibule");
                char part = fs->volume[fcb_address + j];
                file_name[j] = part;
        }
        printf("%s %d\n", file_name,file_size_list[i]);
}
```

That is the whole thought of mine to implement the "$fs\_gsys(LS\_S)\ operation$".

I have finished my discussion about my thought of design for all the parts of this assignment.

## 2 The problems I met in this assignment and the solutions of them:

(1) When I try to update the "$fcb\ part$", I find that if we only update it in the write or read function, we cannot get the file name. For example, when the file is first written into the file system, if we only update the "$fcb\ part$" in the write function, we cannot get the file name because the file name "$s$" is in the open function parameters but not in the write function parameters which makes it impossible to

update the "$fcb\ part$". After observation, I have found that we only need to update the file name when we first write a new file into the file system. Therefore, I update the file name part of the "$fcb\ part$" in the open function. Finally, this problem is solved.

```
u32 length = 0;
while (true) {
        if (s[length] != '\0') {
                length += 1;
        }
        else {
                break;
        }
}
for (u32 i = 0; i < length; i++) {
        fs->volume[fcb_block_start + i] = s[i];
}
update_fcb(fs, fcb_block_start, new_addr, 0, 1);
```

(2) When I write my own function "$update\_fcb$" to update information in the "$fcb\ part$". I find that some information does not need to update in some cases. For example, when we call the "$update\_fcb$" function in the write function, we do not need to update the create time of the corresponding file. This problem makes the update function not universal. After observation, I have found that there are only two cases to use the update function (for the RM operation, I can delete the information in the RM function). One is to update the create time and the other is to update the modified time. The other information left to update is similar and can be controlled. Therefore, I add a parameter called "$case\_number$" to judge the case is which one and then run the corresponding functions.

```
if (case_number == 1) {
        int create_time_RSB_1 = created_time - ((created_time >> 8) << 8);
        int create_time_RSB_2= ((created_time - ((created_time >> 16) << 16) - create_time_RSB_1) >> 8);
        fs->volume[fcb_block_start + 28 + 0] = create_time_RSB_1;
        fs->volume[fcb_block_start + 28 + 1] = create_time_RSB_2;
        fs->volume[fcb_block_start + 30 + 0] = 0;
        fs->volume[fcb_block_start + 30 + 1] = 0;
}

if (case_number == 2) {
        int modified_time_RSB_1= modified_time - ((modified_time >> 8) << 8);
        int modified_time_RSB_2 = ((modified_time - ((modified_time >> 16) << 16) - modified_time_RSB_1) >> 8);
        fs->volume[fcb_block_start + 30 + 0] = modified_time_RSB_1;
        fs->volume[fcb_block_start + 30 + 1] = modified_time_RSB_2;
}
```

Finally, this problem is solved.

(3) When I try to manage the "$fcb\ part$". I find that only one byte for address, size and other information are not enough. Therefore, I finally choose a "$20 + 4 + 4 + 2 + 2$" plan. For every 32 bytes block in the "$fcb\ part$". I use 20 bytes to store the file name, 4 bytes to store the file address, 4 bytes to store the file size and 2 bytes to store the create time of the file and 2 bytes to store the modified time of the file. And I use the bitwise operation to divide the u32 into parts. For example:

```
int address_RSB_1 = file_address - ((file_address >> 8) << 8);
int address_RSB_2 = ((file_address - ((file_address >> 16) << 16) - address_RSB_1) >> 8);
int address_RSB_3 = ((file_address - ((file_address >> 24) << 24) - address_RSB_1 - (address_RSB_2 << 8)) >> 16);
int address_RSB_4 = file_address >> 24;

fs->volume[fcb_block_start + 20 + 0] = address_RSB_1;
fs->volume[fcb_block_start + 20 + 1] = address_RSB_2;
fs->volume[fcb_block_start + 20 + 2] = address_RSB_3;
fs->volume[fcb_block_start + 20 + 3] = address_RSB_4;
```

Finally, this problem is solved.

(4) When I implement the "compact", I find that because the contents of files are stored blocks by blocks but not byte by byte, we cannot stop the "compact" process by reading byte by byte until reaching a NULL. Therefore, after observation, I decide to compact from the end of the file to the end of the "storage part" using overloading and change the old contents to NULL. For example:

```
u32 file_address = get_address_from_FCB(fs, fp);
u32 file_address_2 = file_address + decrease_block_number * 32;
u32 compact_time = 1085440 - file_address_2;
for (u32 i = 0; i < compact_time; i++) {
        fs->volume[file_address + i] = fs->volume[file_address_2 + i];
        fs->volume[file_address_2 + i] = NULL;
}
```

Finally, this problem is solved.

## 3 The steps to execute the program:

The execution steps of running the program in corresponding environment are shown as follows:

(1) Launch VS2017 and set the environment for CUDA programming

(2) Click "$File \rightarrow Open \rightarrow Project/Solution$" in order and then find the "CSC3150_A3.sln" and click it

(3) Make sure that the five files ("main.cu", "file_system.h", "file_system.cu", "user_program.cu" and "data.bin") are in the same folder "CSC3150_A4"

(4) Use "Ctrl+F7" to compile the program

(5) Use "Ctrl+F5" to run the program

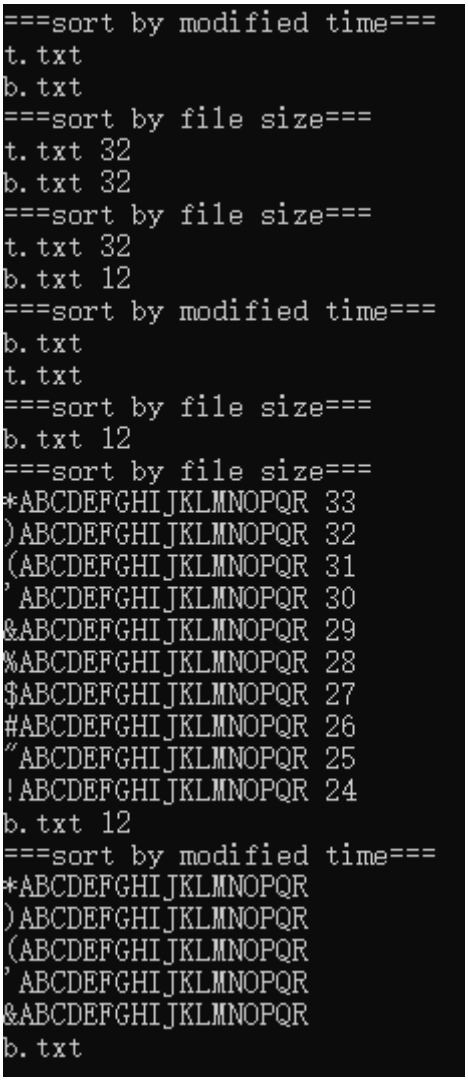
(6) Wait until the program finished

(7) Then you will see the output in the terminal and there will form a "snapshot.bin" in the folder.

## 4 The screenshot of the program output:

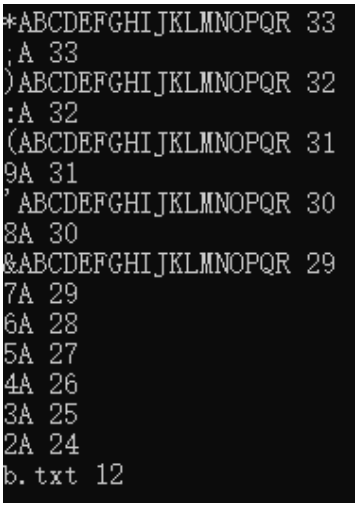
Test case 1:

Test case 2:




Test case 3:

Because the test case 3 is too long, I only cut the end part of it to show:



## 5 Bonus

In this bonus part, I implement the "$fs\_gsy(fs, MKDIR, "app\0")\ operation$" to create a directory named "app":

```
else if (op == MKDIR) {
        //we will create a folder in the file system
        u32 fcb_block_start = 0;
        for (u32 i = 0; i < fs->FCB_ENTRIES; i++) {
                fcb_block_start = fs->SUPERBLOCK_SIZE + i * fs->FCB_SIZE;
                if (fs->volume[fcb_block_start] != NULL) {
                        continue;
                }
                else {
                        created_time += 1;
                        u32 used_block_number = get_block_used_number(fs);
                        u32 new_addr = fs->FILE_BASE_ADDRESS + used_block_number * 32;
                        u32 length = 0;
                        while (true) {
                                if (s[length] != '\0') {
                                        length += 1;
                                }
                                else {
                                        break;
                                }
                        }
                        for (u32 i = 0; i < length; i++) {
                                fs->volume[fcb_block_start + i] = s[i];
                        }
                        update_fcb(fs, fcb_block_start, new_addr, 0, 1);
                        break;
                }
        }
}
```

I implement the "$fs\_gsy(fs, RM\_RF, "app\0")\ operation$" to remove the app directory and all its subdirectories and files recursively.

```
else if (op == RM_RF) {
        if (fp == -1) {
                printf("there is no such directory so you cannot remove it\n");
        }
        else {
                //update the super block
                u32 directory_size = get_size_from_FCB(fs, fp);
                u32 decrease_block_number = directory_size / 32 + 1;
                u32 used_block_number = get_block_used_number(fs);
                used_block_number -= decrease_block_number;
                update_super_block(fs, used_block_number);
                //update the storage
                u32 directory_address = get_address_from_FCB(fs, fp);
                u32 directory_address_2 = directory_address + decrease_block_number * 32;
                u32 compact_time = 1085440 - directory_address_2;
                for (u32 i = 0; i < compact_time; i++) {
                        fs->volume[directory_address + i] = fs->volume[directory_address_2 + i];
                        fs->volume[directory_address_2 + i] = NULL;
                }
                //update the fcb
                for (u32 i = 0; i < 32; i++) {
                        fs->volume[fp + i] = NULL;
                }
        }
}
```

I implement the "$fs\_gsys(fs, PWD)$ $operation$" to print the path name of current:

```
else if (op == PWD) {
u32 file_name_fp_list[1024];
u32 index = 0;
for (u32 i = 0; i < 1024; i++) {
        u32 fcb_block_start = fs->SUPERBLOCK_SIZE + i * fs->FCB_SIZE;
        if (fs->volume[fcb_block_start] != NULL) {
                file_name_fp_list[index] = fcb_block_start;
                index += 1;
        }
}
for (u32 i = 0; i < index; i++) {
        u32 fcb_address = file_name_fp_list[i];
        char file_name[20];
        for (u32 j = 0; j < 20; j++) {
                char part = fs->volume[fcb_address + j];
                file_name[j] = part;
        }
        printf("D:\Bonus\CSC3150_A4_Bonus\CSC3150_A4_Bonus\%s\n", file_name);
}
}
```

I also implement the "$fs\_gsy(fs, LS\_D/LS\_S)$ $operation$" to list the files as well as directories. For example (use $LS\_D$ as an example):

```
if (op == LS_D) {
        printf("===sort by modified time===\n");
        u32 file_name_fp_list[1024];
        u32 index = 0;
        for (u32 i = 0; i < 1024; i++) {
                u32 fcb_block_start = fs->SUPERBLOCK_SIZE + i * fs->FCB_SIZE;
                if (fs->volume[fcb_block_start] != NULL) {
                        file_name_fp_list[index] = fcb_block_start;
                        index += 1;
                }
        }
        u32 file_modified_time_list[1024];
        u32 file_modified_time = 0;
        for (u32 i = 0; i < index; i++) {
                file_modified_time = get_modified_time_from_FCB(fs, file_name_fp_list[i]);
                file_modified_time_list[i] = file_modified_time;
        }
        //use bubble sort to sort
        for (u32 i = 0; i < index - 1; i++) {
                for (u32 j = 0; j < index - 1 - i; j++) {
                        if (file_modified_time_list[j] < file_modified_time_list[j + 1]) {
                                u32 temp_1 = file_modified_time_list[j];
                                file_modified_time_list[j] = file_modified_time_list[j + 1];
                                file_modified_time_list[j + 1] = temp_1;
                                u32 temp_2 = file_name_fp_list[j];
                                file_name_fp_list[j] = file_name_fp_list[j + 1];
                                file_name_fp_list[j + 1] = temp_2;
                        }
                }
        }
        //print the file names in order
        for (u32 i = 0; i < index; i++) {
                u32 fcb_address = file_name_fp_list[i];
                char file_name[20];
                for (u32 j = 0; j < 20; j++) {
                        char part = fs->volume[fcb_address + j];
                        file_name[j] = part;
                }
                printf("%s\n", file_name);
        }
}
```

That is all what I have made in the “Bonus Part”.

## 6 What did I learn from this assignment:

In this assignment, I have learned the whole structure of file system and learn how to manage the file system using a lot of parts in it like “super block part” and “$fcb\ part$”. I also learn a lot of operations in the file system like read and write. All in all, I have learnt a lot in this assignment.