



THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC 3150

OPERATING SYSTEMS

Assignment 4

Author:
Shi Wenlan

Student Number:
119010265

November 24, 2021

Contents

1	Task 1	2
1.1	How did I design the program	2
1.2	Problems I met:	4
1.3	The steps to execute my program	5
1.4	Screenshots of my program output	5
1.5	What did I learn from the task	6
2	Bonus	6
2.1	How did I design the program	6
2.2	Problems I met:	7
2.3	The steps to execute my program	8
2.4	Screenshots of my program output	9
2.5	What did I learn from the task	9

1 Task 1

1.1 How did I design the program

How to implement volume structure: A volume is an array of 1085440 unsigned chars. It consists of free space management blocks (4096 unsigned chars), file control blocks (32768 unsigned chars), and contents of files (1048576 unsigned chars).

How to implement free space management: 4096 unsigned chars are composed of 32768 bits, which correspond one-to-one with the 32768 storage blocks of the stored files. When the bit is 1, it means that the block has been used by a file, and when the bit is 0, it means that the block is empty and can be allocated to the file. I wrote the *bit_check()* function to confirm the usage of the blocks to assist in allocating space to the newly written file. I also wrote the *bit_change()* function to modify the value of a specific bit in an unsigned char.

How to implement contiguous allocation: Whenever I need to allocate space for a file, I use the *bit_check()* function in a loop to check free space management blocks, and use *count* to count the number of consecutive empty blocks. When 32 consecutive empty blocks are found, the address of the first block is passed to the file control block.

How to implement file control block: The 32 bytes of the file control block are allocated as follows: 0 to 19 bytes for file name, 20 to 21 bytes for size (bytes, 0 to 1023), 22 to 23 bytes for start address (storage block index, 0 to 32767), 24 to 27 bits for create time, and 28 to 31 bits for modify time. The source of the time value is the global variable *gtime*. Each time the *fs* series function is executed, *gtime* is incremented by 1.

How to implement *fs_open()*: Whenever a file is to be opened, the program will perform the following steps in sequence: first check whether it is a file recorded by the file control block (existing file) according to the file name, if it is, then locate the file control block, if not, Then create a new file control block. For an existing file control block, if the access right is *G_READ*, the address of the file control block is returned, and the program ends. If the access right is *G_WRITE*, then the corresponding free space management blocks is cleared according to the size and start address recorded by the file control block, and then reset the size to zero, then use the *find_space()* function to find a continuous 32 block space, store the address of the first block in the file control block to overwrite the old address, after

that, update the modify time of the file control block, and finally return the address of the file control block, and the program ends. For the newly created file control block, if the access right is *R_READ*, an error is reported and the program ends. If the access right is *R_WRITE*, use the *find_space()* function to find a space of 32 consecutive blocks, record the first address in the file control block, and record the file name, create time, modify time (equals to create time), size (equals to 0) are stored in the file control block, then return the address of the file control block, and the program ends.

How to implement *fs_read()*: The program first reads the *size* stored in the file control block, compares it with the size requested by the user, and takes the shorter one as the size of the final read content. Then read the start address stored in the file control block, and use it to read the content of the required length to the result.

How to implement *fs_write()*: The program first reads the start address stored in the file control block, calculates the number of storage blocks that need to be used, writes the content to its corresponding storage blocks, and sets the corresponding Free Space Management blocks bit to 1, and then update the size stored in the file control block to the number of bytes written, and update the modify time to the current point in time.

How to implement *fs_gsys(LS_D)* and *fs_gsys(LS_S)*: In order to facilitate sorting and output, I created a new structure named *FCB* to store the file control block information, and an array named *min_heap* to store FCB pointers which is used to sort the structure according to the value of a certain attribute. I use min heap to sort FCB pointers. According to the different attributes that sort base on, I wrote three versions of the *perlocateDown* helper function (*perlocateDown_D()* for *fs_gsys(LS_D)*, *perlocateDown_S()* for *fs_gsys(LS_S)*, *perlocateDown_C* for *compaction()*) to help sorting. Finally, output the sorted file control block information during the sorting process.

How to implement *fs_gsys(RM)*: Whenever a file is to be deleted, the program first compares the file control blocks according to the input file name, finds the file control block with the same file name (outputs the error message if it cannot be found), and sets the first part of the storage file name to 0, read its stored start address and size, and use these two information to set the bits of the Free Space Management blocks corresponding to the storage block it occupies to 0.

How to implement *compaction()*: Whenever the fs series function calls the *find_space()* function to obtain the new address, and the space of 32 consecutive

blocks is not found, *find_space()* will return 99999. When the new address obtained by the fs series function is 99999, the fs series function will execute once *compaction()* to exclude external fragmentation, and then call the *find_space()* function again to get the new address. If fs series function still get 99999 this time, it means that the file system is really full. In the *compaction()*, I sort the file control blocks using min heap in the order of start address from small to large, and then moved the storage content corresponding to each file control block to the lower address as much as possible in sequence. The movement is implemented as follows: first set the corresponding bit of the Free Space Management blocks corresponding to the file to 0, and then use the *find_space()* function to find the continuous storage space which is enough to store the content of the file with the smallest start address, and then set the corresponding bit of the Free Space Management blocks to 1, finally update the start address stored in the file control block.

1.2 Problems I met:

Part of Output information being swallowed by VS: When testing the third test case, the VS output information is incomplete or even misplaced, but the TC301 computer can display the output completely and correctly. I changed the output solution and solved it: Originally, when reading the file name from the file control block, every time a char is read, a char is printed in the form of %c. Now a local array is built to store the read file name, in the form of %s, then printf the entire array.

File name matching error: When testing the third test case, I found that files with the same prefix and different full names are considered the same file. After checking the code, it was found that there was a problem with the while loop condition setting at the time of file identification, and it was solved after modification: Originally, name stored in the file control block and the name entered by the user in turn read the char and continue the loop when they are both not 0, now, the loop will be terminated when the chars read are both 0.

Confusion between storage block index and file control block index: When testing the first test case, it was found that the size of *b.txt* was changed unexpectedly when reading and writing the *t.txt* file. After checking the code, it was found that the storage block index and the file control block index were confused when writing the code. If the fp sent by *fs_open* is the address of the file storage blocks location, the file control block address cannot be deduced to update the file information during the read and write phase. I modified the content transfered by to solve the problem: Originally, the storage block index used as the file control block

index in my code resulted in the modification of the unexpected file control block, now, the returned fp stores the address of the file control block. The subsequent read and write steps obtain file information through the file control block, and read and write in the designated storage block.

1.3 The steps to execute my program

Windows (using Visual Studio): Add existing items (the source codes) into your cuda project. Click to compile each .cu file. Then run the program.

Linux (using .sh script): Get into the source code folder in the terminal. Then enter `/bin/shscript.sh` to run the script.

1.4 Screenshots of my program output

Figure 1: Screenshots of 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==
t.txt
b.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
E:\CUDA 11.1 Runtime\Release\CUDA 11.1 Runtime1.exe (进程 8) 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 2: Screenshots of test case 2

```

Microsoft Visual Studio 调试控制台
BA 42
CA 41
BA 40
AA 39
AA 38
CA 37
A 36
A 35
A 34
*ABCEFGHIJKLMNOPQR 33
A 33
ABCEFGHIJKLMNOPQR 32
A 32
ABCEFGHIJKLMNOPQR 31
BA 31
ABCEFGHIJKLMNOPQR 30
BA 30
ABCEFGHIJKLMNOPQR 29
A 29
BA 28
BA 27
AA 26
BA 25
BA 24
b.txt 12
E:\CUDA 11.1 Runtime\Release\CUDA 11.1 Runtime1.exe (进程 21812) 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 3: Screenshots of test case 3

1.5 What did I learn from the task

I have a deep understanding of the theory and implementation of a single-level directory structure file system. I have also exercised my ability to output debugging.

2 Bonus

2.1 How did I design the program

How to implement directory: I regard a directory as a special file. The only difference between its file control block and that of ordinary files is that the first

bit of the 23rd byte is 1. Whenever I want to open or delete a file or directory, I can check whether the owner of the current file control block is a directory or a file by checking the first bit of the 23rd byte. I store the names of all files and subdirectories in the directory in the directory file, and the total length is the size of the directory file. Whenever I need to delete or create a new file or directory in the current directory, I must remember to update Free Space Management blocks, the size and modify time parameters of file control block, and the stored content of the current directory. Taking into account that the directory (a special file) needs to be updated repeatedly in supplementary form, it is inconvenient to allocate new space each time. Therefore, every time a new directory (a special file) is created, 32 blocks of space are fixed to it. And the corresponding bits in Free Space Management blocks will all be set to 1 at one time until the directory is deleted. So I don't need to update Free Space Management blocks every time.

How to implement multi-level directory structure: Considering that neither the file control block nor the storage block has room to store pointers to the doubly linked list, I additionally set the global variables *pwd* (u32 array, used to store the current path) and *pt* (int, indicating the current directory level). The address of the file control block of the directory is stored in *pwd*.

How to open an existing file in the current directory: My original idea was to read the contents of the file in the current directory first, check whether the file name exists in it, if it exists, traverse the file control blocks to find the file control block of the target file by comparing the file name, if it does not exist, it means that the file requested to open is not exists in the current directory. But this gave rise to the third problem I encountered (explained later). After adding the global variable back to solve the problem, my idea becomes to traverse all file control blocks directly, find the file control block whose file name can be matched, and the file control block address stored in the corresponding *back* is equal to the file control block address of the current directory.

2.2 Problems I met:

The program terminates abnormally during initialization: In order to facilitate the management of the hierarchical structure of files, I created a root directory named *root* when the file system was initialized, and all new files and directories to be made in the future will be in the root directory. But during the test, it was found that the program crashed without completing the initialization. After checking the code, it was found that the root directory did not have a parent directory, and the MKDIR function used for general directories was not applicable for root directory,

which caused the array out-of-bounds and program crashed. After that, I wrote a customized initialization code for the root directory to solve the problem.

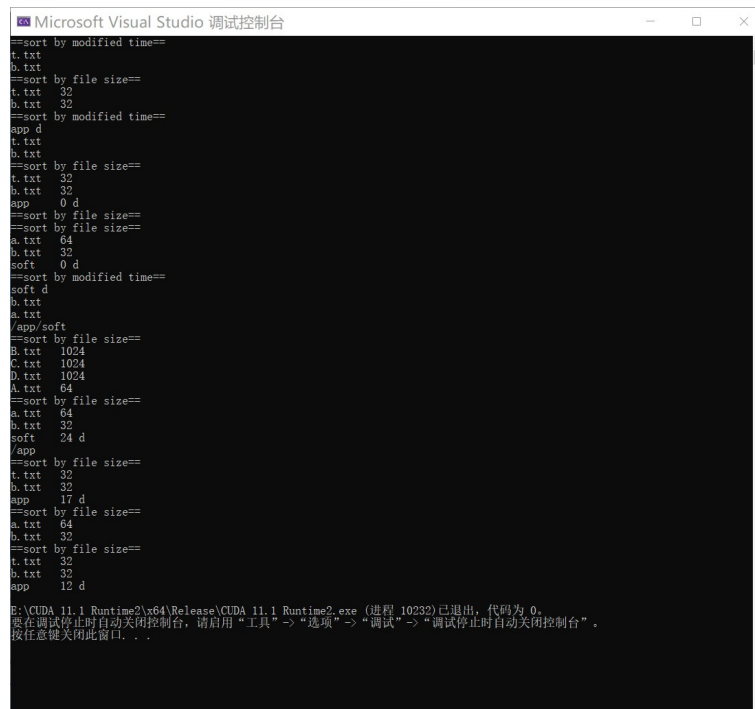
Cuda does not support recursive functions: After fixing the above bug, the program will still crash where it has been confirmed that there is no problem after the initialization is completed. After checking the information on the Internet, I learned that Cuda does not support recursive functions. After operations and attempts, I confirmed that Cuda does not even support function calls with the same name but different branches. As long as this situation exists, the program will crash midway. After that, I changed the recursive part of the file with the three-layer of same code, and replaced the other similar recursive but non-recursive parts with helper functions with the same content, then solve the problem.

Files with the same name cannot be created in different directories: Because the file control block of the corresponding file is searched by comparing the file name, the file control block of the file with the same name in different folders will be confused with each other. In order to distinguish them, I created a new global variable named *back* (an array of 1024 u32), which corresponds to the file control blocks one-to-one, and stores the address of the file control block corresponding to the directory where the file control block is located. Each time a file or directory is created or deleted, the *back* is updated at the same time, and the file control block of the files in the current folder is filtered through the *back* when the file is searched. I solved the problem in this way.

2.3 The steps to execute my program

The same as that in Task1 part.

2.4 Screenshots of my program output



```

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 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 24 d
/app
==sort by file size==
t.txt 32
b.txt 32
app 17 d
==sort by file size==
a.txt 64
b.txt 32
==sort by file size==
t.txt 32
b.txt 32
app 12 d
E:\CUDA 11.1 Runtime2\64\Release\CUDA 11.1 Runtime2.exe (进程 10232) 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 4: Screenshots of bonus test case

2.5 What did I learn from the task

I have a deep understanding of the theory and implementation of a multi-level directory structure file system. And I have improved my proficiency in the use of CUDA coding.