CSC3150 Operating Systems

Project 4 report

Name: Xiao Nan

Student ID: 119010344

Date: 2021/11/23

The Chinese University of Hong Kong, Shen Zhen

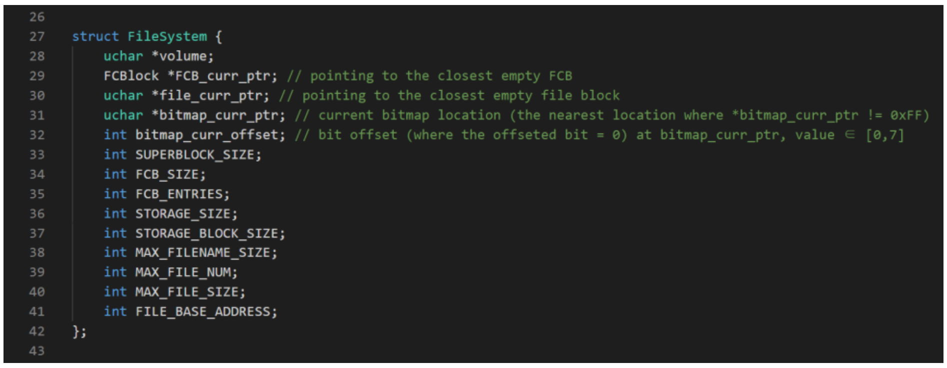
0. Introduction:

In homework 4 basic part I implemented a mini-file system using CUDA GPU card. The mini-file system can support basic file operations like opening, reading, writing, listing and removing files. In order to achieve the above logical operations, I designed the 32-Byte FCB structure to record file information (size, modify time, create time, etc), as well as a 4-Byte superblock for free space management. In the bonus part I designed and implemented tree-structured directory for storing layers of files, as well as commands for managing those files (removing, changing directory, etc).

1. Design of the program:

In the normal part, the mini-file system I designed for homework 4 basic part can be divided into the following sub-parts: Bitmap free space management part, File Control Block (FCB) part, as well as file space part. The functions implemented above them includes open, read/write, listing (LS), as well as remove.

The bitmap free space management part consists of a 4KB block in the global memory, in which each bit is mapped to a 32-Byte block in the file space. The bit is set to 1 when the mapped file block is captured and is set to 0 when the block is free. I management the bitmap zone by maintaining bitmap_curr_ptr in fs, a pointer always pointing to the last 8-bit region where there is at least one bit is 0 (indicating there is free space mapped) inside the region. I also maintained bitmap_curr_offset, recording the free space bit that is closest to the beginning of the bitmap zone. Whenever a file is opened or written in more bytes, free space bits would be retrieved from bitmap_curr_ptr with the help of bitmap_curr_offset and then be set to 1. Similarly when the file is removed or written in fewer bytes the bits would be set to 0. After the bit operations are done, bitmap_curr_ptr and bitmap_curr_offset would be updated to the most recent free space bit for the convenience of the next time's usage. The designs of the pointers managing those regions stored in file system is shown in figure 1.1.

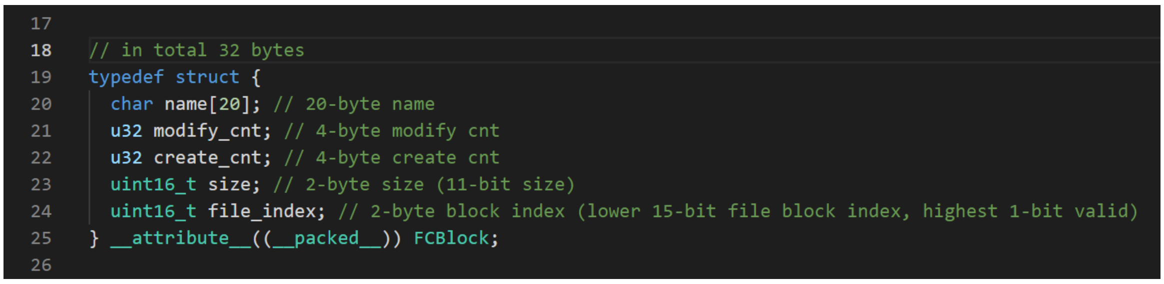


```

struct FileSystem {
    uchar *volume;
    FCBlock *FCB_curr_ptr; // pointing to the closest empty FCB
    uchar *file_curr_ptr; // pointing to the closest empty file block
    uchar *bitmap_curr_ptr; // current bitmap location (the nearest location where *bitmap_curr_ptr != 0xFF)
    int bitmap_curr_offset; // bit offset (where the offseted bit = 0) at bitmap_curr_ptr, value ∈ [0,7]
    int SUPERBLOCK_SIZE;
    int FCB_SIZE;
    int FCB_ENTRIES;
    int STORAGE_SIZE;
    int STORAGE_BLOCK_SIZE;
    int MAX_FILENAME_SIZE;
    int MAX_FILE_NUM;
    int MAX_FILE_SIZE;
    int FILE_BASE_ADDRESS;
};

```

Figure 1.1: pointers in fs managing the memory regions

In addition to the bitmap region, the file space itself maintains a compact structure, which saves space and improves the performance of looking up the files physically. All files in the file space aligned in blocks are tightly placed together in order to reduce external fragmentation and allow further allocations of large files. The compact structure is maintained by moving the latter files further when the target file is written in more bytes, and moving those files closer when the target file is written in fewer bytes or removed. File_curr_ptr, a pointer pointing to the empty location of the file space that is most closed to the file space starting address, is maintained in order to facilitate compacting the files back and forth. Such pointer is updated whenever an operation on file space which requires extending or shrinking the number of blocks used is carried out.

In order to manage the metadata of the files in the file space, I designed File Control Block to be a 32-Byte struct (in C) consisting of the following elements (as shown in figure 1.2): A 20-Byte file name array storing the file name. A 4-byte modify count indicating the latest modified time of the file. A 4-byte create count recording the created time of the file. A 2-byte size count for storing the size of the file (when opening, the file size is 0 in default). A 2-byte file_index indicating the location of the file in the file space, inside which the lowest 15 bits are used to store the file block index (indicating the starting block of the file in the file space) and the highest 1 bit is used as the valid bit (indicating whether the FCB is valid, i.e., used by a file). The FCB also maintains a compact structure, in which when removing a file and its corresponding FCB being removed, the file system would move the latter files in the FCB space back in order to maintain the compact structure. In addition, a FCB_curr_ptr pointing to the closest free space is maintained in the file system for the convenience of compacting the FCBs.



```

// in total 32 bytes
typedef struct {
  char name[20]; // 20-byte name
  u32 modify_cnt; // 4-byte modify cnt
  u32 create_cnt; // 4-byte create cnt
  uint16_t size; // 2-byte size (11-bit size)
  uint16_t file_index; // 2-byte block index (lower 15-bit file block index, highest 1-bit valid)
} __attribute__((__packed__)) FCBlock;

```

Figure 1.2: FCB for the normal part

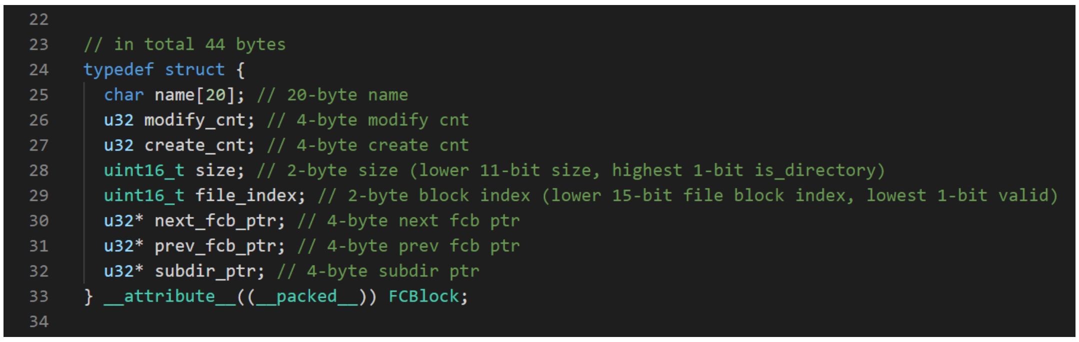
Based on the above structural designs, I implemented read/write, remove and list functions.

For reading, if the file is opened with G_READ (read mode), then the program would simply read the contents character by character to the output file, otherwise an error message would be printed out. For writing, if the program is in the writing mode (G_WRITE), it would first determine the number of blocks to scale (block_size_diff): If that is zero then the program simply writes to the file space. If that is greater than zero the program would do compaction by moving the latter files further according to the block_size_diff and then extend the current file blocks, with the bitmap being set 1 in corresponding regions by using bitmap_curr_ptr and bitmap_curr_offset. If that is less than zero then the current file block number would shrink and the latter files would be moved back to maintain the compact structure, with the bitmap being set 0 in corresponding regions. In such compacting scenario the file_curr_ptr helps identifying the compact ending location. After compacting the FCB information (such as the size and the file_index) would be updated and the file pointer (indicating the file space index) would be returned.

For removing, the program would first judge the number of blocks to remove. Then it removes all the blocks and FCB of the file, compact the latter files or FCBs in both the file space and the FCB space, and then update the current FCB pointers and file space pointers in the file system.

The listing function is divided into 2 categories: LS_S and LS_D. In LS_S the files are listed out in the order of file size, together with the files of the same size listed according to their created time (first create first list). LS_D lists out the file according to their last modify time. The two listing functions are implemented similarly by first preparing the pointers to the files and then perform a quicksort (according to size or modify time) in order to get the sorted list of pointers. Then the results are printed out by those sorted pointers.

In the bonus part, I treat the directory as a special kind of file whose information are stored in the FCB too. the FCB (indicated in figure 1.3) is extended into 44 bytes per entry. Each FCB contains the file name (20 bytes), modify count (4 bytes) which is updated whenever a file or directory inside is created or modified, create count (4 bytes) which is updated whenever the file/directory is created, size (4 bytes), file index (4 bytes), as well as the pointers used for the linked list structure implementation of the logically tree-structured directory: The next_fcb_ptr (4 bytes) points to the next FCB under the current directory (which is 0 if the current FCB indicates the last file in the directory). The prev_fcb_ptr (4 bytes) points to the previous FCB under the current directory (which is 0 when the current FCB indicates the first file in the directory). The subdir_ptr (4 bytes) points to the files/directories in the sub-directory, which is used only when the current FCB indicates a sub-directory.



```
// in total 44 bytes
typedef struct {
  char name[20]; // 20-byte name
  u32 modify_cnt; // 4-byte modify cnt
  u32 create_cnt; // 4-byte create cnt
  uint16_t size; // 2-byte size (lower 11-bit size, highest 1-bit is_directory)
  uint16_t file_index; // 2-byte block index (lower 15-bit file block index, lowest 1-bit valid)
  u32* next_fcb_ptr; // 4-byte next fcb ptr
  u32* prev_fcb_ptr; // 4-byte prev fcb ptr
  u32* subdir_ptr; // 4-byte subdir ptr
} __attribute__((__packed__)) FCBlock;
```

Figure 1.3: Bonus part's FCB structure

In order to track the current directory and parent directory, I used several __device__ global variables to track the directory pointers: FCBlock* curr_dir for keeping track of the pointer to the

FCB of the current directory, FCBlock* par_dir for keeping track of the pointer to the FCB of the parent directory (which is set 0 if the current directory is the root directory). FCBlock* pwd[4] for the path to the current directory (which may not be fully used depending on the level of directory currently the user is in, and makes changing directory easy by assigning the directory to be changed to the array).

The functions of the file system is divided into opening files, reading/writing files, listing files, removing files, creating directory, changing directory, removing files recursively (RM_RF). When opening a file the program would first allocate file space and FCB space as well as the bitmap in the same procedure as that is performed in the normal part, except that the logical structure of the file under the current directory, a linked list of files or sub-directories under the current directory, is updated. Similarly when performing operations requiring compaction, including removing files/sub-directories, the logical structures (the next/previous pointers) of the FCB would be updated because of the modification of the physical location of the FCBs. When listing files, instead of Brute-force listing all files in the FCB region, the program would traverse through the curr_dir pointer, sort those pointers using quicksort, and then list out the sorted results in the order of modification time or size (with first create first list implemented when some files are having the same size). When creating directory the program would do similar procedures as creating a file, except that it needn’t allocate file space and only FCB space and bitmap space are allocated. When executing RM_RF the program would use depth first search (dfs_rm()) to seek all files inside a directory. If the function comes across a file or an empty directory it would remove that directly. If the function comes across a non-empty directory it would recursively remove all files inside and finally remove the remaining empty directory.

2. Runtime environment:

2.1. Linux version:

```
[cuhksz@TC-301-37 xxx]$ cat /proc/version
Linux version 3.10.0-957.21.3.el7.x86_64 (mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623 (Red Hat 4.8.5-36) (GCC) ) #1 SMP Tue Jun 18 16:35:19 UTC 2019
```

2.2. CUDA version:

```
[cuhksz@TC-301-37 xxx]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Fri_Feb__8_19:08:17_PST_2019
Cuda compilation tools, release 10.1, V10.1.105
```

2.3. GPU information:

```
[cuhksz@TC-301-37 xxx]$ lshw -C display
WARNING: you should run this program as super-user.
  *-display
       description: VGA compatible controller
       product: GP106 [GeForce GTX 1060 6GB]
       vendor: NVIDIA Corporation
       physical id: 0
       bus info: pci@0000:01:00.0
       version: a1
       width: 64 bits
       clock: 33MHz
       capabilities: vga_controller bus_master cap_list rom
       configuration: driver=nvidia latency=0
       resources: irq:142 memory:f0000000-f0ffffff memory:c0000000-cfffffff memory:d0000000-d1ffffff ioport:3000(size=128) memory:f1080000-f10fffff
  *-display
       description: VGA compatible controller
       product: UHD Graphics 630 (Desktop)
```

```
          vendor: Intel Corporation
          physical id: 2
          bus info: pci@0000:00:02.0
          version: 00
          width: 64 bits
          clock: 33MHz
          capabilities: vga_controller bus_master cap_list rom
          configuration: driver=i915 latency=0
          resources: iomemory:400-3ff irq:140 memory:4000000000-4000ffffff memory:e0000000-efffffff ioport:4000(size=64)
WARNING: output may be incomplete or inaccurate, you should run this program as super-user.
```

3. Steps to execute my program:

In command line, you need to first unzip the .zip file. Make sure that the .zip file is under the current directory:

```
[cuhksz@TC-301-37 xxx]$ unzip Assignment_4_119010344.zip
```

3.1 Normal part:

“cd” to the “Source” execution directory, and type “make” to compile the source code. Make sure that the main.cu, user_program.cu, file_system.cu, file_system.h are under the current directory:

```
[cuhksz@TC-301-37 xxx]$ cd Assignment_4_119010344/Source/
[cuhksz@TC-301-37 Source]$ make
```

Then, type “./main.out” to execute the program.

```
[cuhksz@TC-301-37 Source]$ ./main.out
```

3.2 Bonus program:

“cd” to the “Bonus” execution directory. Type “make” to compile the bonus programs. Make sure that the main.cu, user_program.cu, file_system.cu, file_system.h are under the current directory:

```
[cuhksz@TC-301-37 xxx]$ cd Assignment_4_119010344/Bonus/
[cuhksz@TC-301-37 Bonus]$ make
```

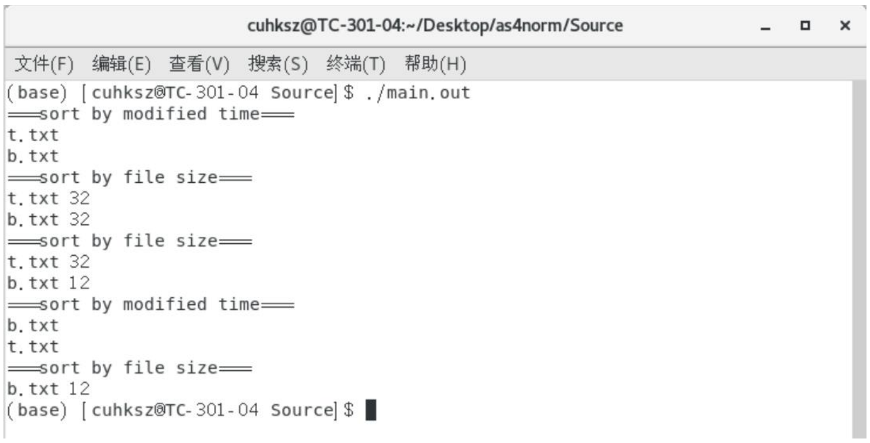
Then execute the bonus program:

```
[cuhksz@TC-301-37 Bonus]$ ./main.out
```

4. Program output screenshots:

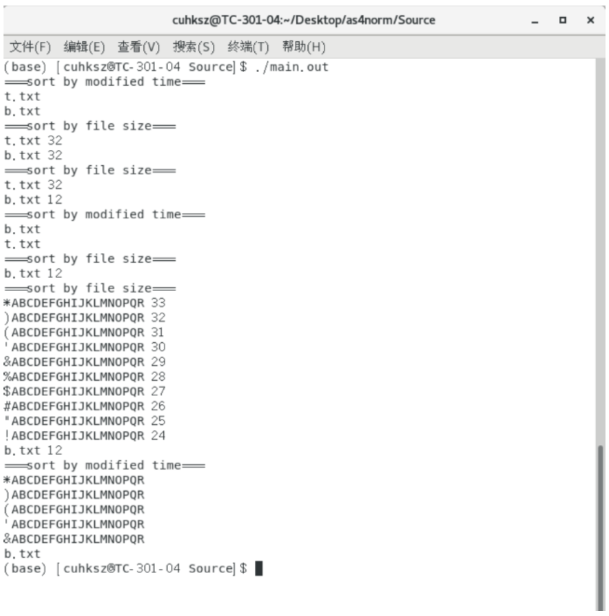
4.1 Normal program:

The results for the test cases 1, 2, 3 are shown in the figure 4.1, 4.2, 4.3, respectively:


cuhksz@TC-301-04:~/Desktop/as4norm/Source
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

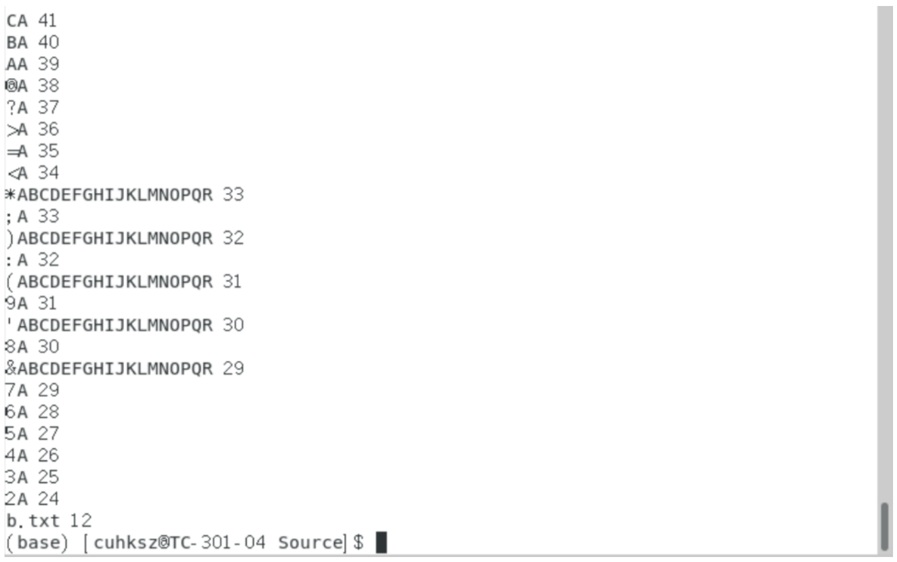
```
(base) [cuhksz@TC-301-04 Source]$ ./main.out
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by file size===
t.txt 32
b.txt 12
===sort by modified time===
b.txt
t.txt
===sort by file size===
b.txt 12
(base) [cuhksz@TC-301-04 Source]$ 
```

Figure 4.1: Test case 1 results


cuhksz@TC-301-04:~/Desktop/as4norm/Source
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

```
(base) [cuhksz@TC-301-04 Source]$ ./main.out
===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
$ABCDEFGHIJKLMNOPQR 27
#ABCDEFGHIJKLMNOPQR 26
"ABCDEFGHIJKLMNOPQR 25
!ABCDEFGHIJKLMNOPQR 24
b.txt 12
===sort by modified time===
*ABCDEFGHIJKLMNOPQR
)ABCDEFGHIJKLMNOPQR
(ABCDEFGHIJKLMNOPQR
'ABCDEFGHIJKLMNOPQR
&ABCDEFGHIJKLMNOPQR
b.txt
(base) [cuhksz@TC-301-04 Source]$ 
```

Figure 4.2: Test case 2 results

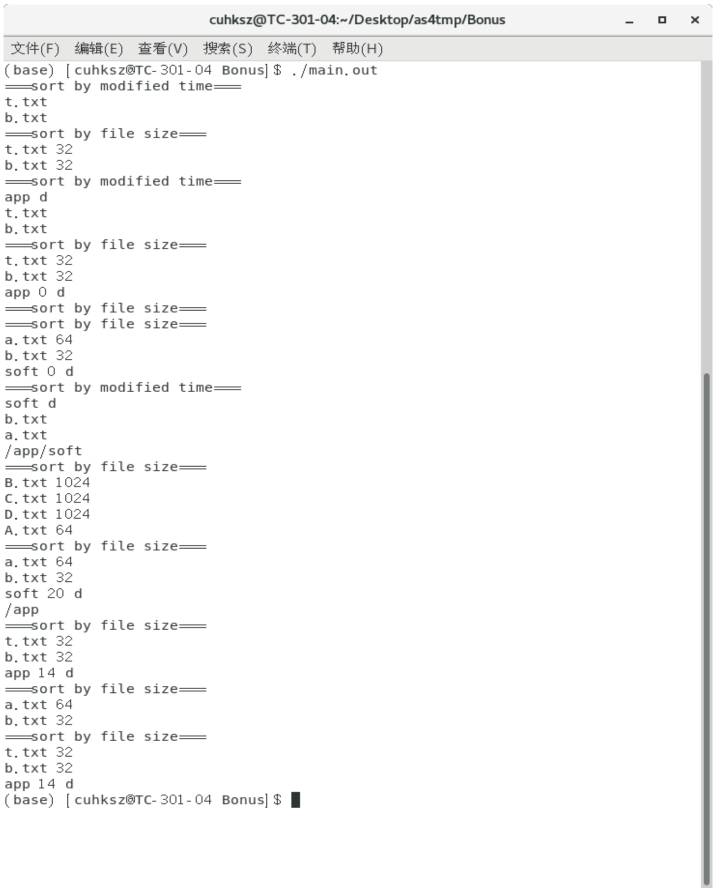


```
CA 41
BA 40
AA 39
@A 38
?A 37
>A 36
=A 35
<A 34
*ABCDEFGHIJKLMNOPQR 33
;A 33
)ABCDEFGHIJKLMNOPQR 32
:A 32
(ABCDEFGHIJKLMNOPQR 31
9A 31
'ABCDEFGHIJKLMNOPQR 30
8A 30
&ABCDEFGHIJKLMNOPQR 29
7A 29
6A 28
5A 27
4A 26
3A 25
2A 24
b.txt 12
(base) [cuhksz@TC-301-04 Source]$
```

Figure 4.3: Part of test case 3 results

4.2 Bonus:

The results for the bonus test case is shown in the figure 4.4:



```
(base) [cuhksz@TC-301-04 Bonus]$ ./main.out
===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 20 d
/app
===sort by file size===
t.txt 32
b.txt 32
app 14 d
===sort by file size===
a.txt 64
b.txt 32
===sort by file size===
t.txt 32
b.txt 32
app 14 d
(base) [cuhksz@TC-301-04 Bonus]$
```

Figure 4.3: Bonus test case results

5. Problems that I've met and how I solved them:

In this project the first problem that I came across is a logical bug which costs me a lot of time to figure out. When I was debugging my bonus program I found out that the program aborts before the removals finish, and when removing files using RM_RF the program always forget removing some of them, where the linked list structure of those FCBs seemed to fail to maintain as the removal process goes. I thought a lot (about 6 hours and above), printed at many places in the program and finally figured out that I made an error when updating the linked list structure under the edge cases: When trying modifying the next pointer of the previous FCB to the next pointer of the current FCB (ptr->prev->next = ptr->next) I failed to include the case where the previous FCB is the current directory (whose pointer to the next file, when the current directory is the directory specified in that FCB, should be referenced using subdir_ptr, meaning that the assignment sentence should be ptr->prev->subdir_ptr = ptr->next and that the previous changing of ptr->prev->next would make the information of the next FCB of the current directory (in the parent directory) lost and hence cause a pointer reference error when the next time the program is trying to dereference it. I now understand that the edge cases, exactly the cases that might occur at the very beginning of the test (where everything is empty in the memory and edge cases happen easily), are most "aggressive" to the program and that I should try thinking carefully enough to deal with those edge cases when I am designing the program structure at the beginning.

Another problem is that I forgot considering the create time of the file at the beginning, which costed me great efforts to redesign the FCB and restructure the program. It inspires me that I should design the key structures of the program carefully before writing (otherwise restructuring the program takes great efforts, especially for large programs with 600+ lines).

6. Things I've learnt from the project:

In this project I learnt deeply how the file system works. I understood how the physical compact structure of the files in the file space can help the operating system save spaces in the secondary storage. I learnt how the file system may open, read/write, remove a file stored in the disk. I also learnt how the operating system can manage the metadata of the files using FCBs and understood that a good design of FCB (with the accuracy of enabling bitwise operations within limited space) can enable more comprehensive descriptions of the files using very few spaces. From the bonus problem, I learnt that the logical tree-structured directory constructed upon the FCBs can help make the files organized in order and provide users with an acceptable way to retrieve data from those files stored in levelled directories. I understand deeply how the logical concepts (directories, files) can be implemented in the secondary storage (in FCBs, free space management region, file space, etc) to facilitate operations on data in the backing store.