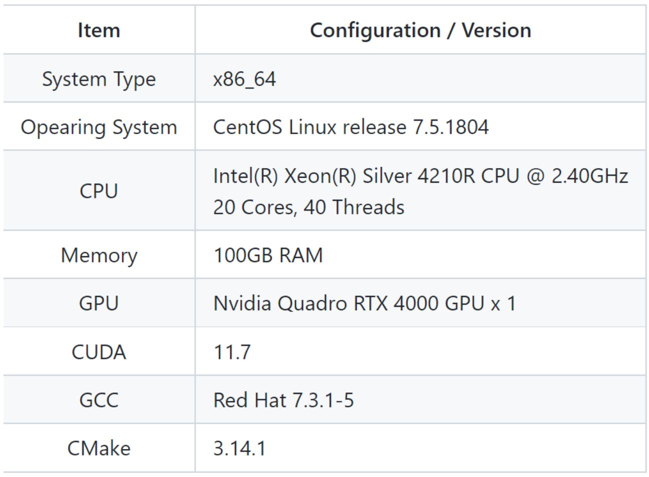
Report

Zhenyu Pan 120090196

November 25, 2022

1 Environment

Since we are recommended to use the cluster to compile and test our codes, I use the cluster provided, thanks for Prof.Chung and TAs bringing the cluster into use. Below is the environment information provided in "Cluster User Guide".



Item	Configuration / Version
System Type	x86_64
Opearing System	CentOS Linux release 7.5.1804
CPU	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz 20 Cores, 40 Threads
Memory	100GB RAM
GPU	Nvidia Quadro RTX 4000 GPU x 1
CUDA	11.7
GCC	Red Hat 7.3.1-5
CMake	3.14.1

Figure 1: Environment Information

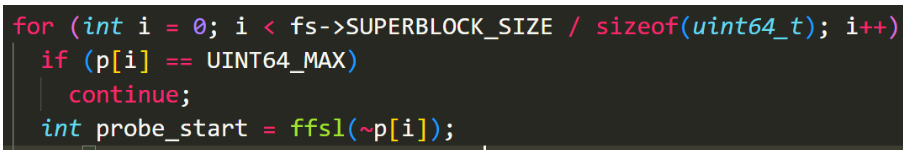
2 Execution Steps

1. Please make sure the script and the corresponding files are in the **same folder**
2. Log in to cluser and enter the folder(eg: cd source/bonus)
3. Type "sbatch ./slurm.sh" in the terminal
4. Wait a few seconds, the task will be done

3 Design of Program

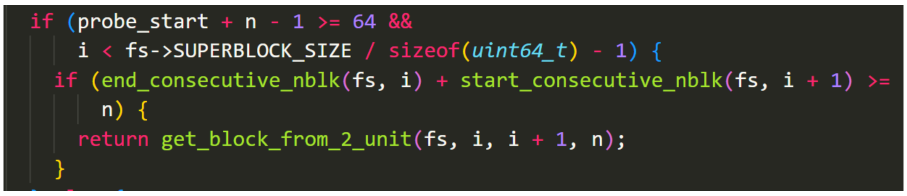
First talk about *fs_open()*: Go through all the files to check if there is a file with the same file name, if there is, return the corresponding file descriptor which is the corresponding array index i; if none, create a new file with the given name. To create a file, we need to find a free file control block(below use FCB to indicate this term). The strategy here is to find the first FCB, which is the logic corresponding to *first_empty_file_pos*. Then the problem now is how to tell if a FCB is available or not, my strategy is to check the file name; if the length of the file name is 0, it means the corresponding FCB is available, not vice versa (Note that in bonus part, due to the the concept of directory, only files under the directory can be traversed to check if there are files with the same name, because files with the same name can exist under different directories). Then *fs_read()*, it is such an easy part: find the corresponding FCB according to the fp, and then calculate the corresponding address of the storage according to the address of the storage stored in FCB. Lastly, just read the content. Then come to a little more complicated part, *fs_write()*: overwrite the contents of a file if existed, consider two cases, **1.** The number of blocks occupied by the new contents is not equal to that of the old contents. A block is 32 bytes, the number of blocks is $\frac{size}{32}$. If the new one is larger than the old one, then the old space is not big enough to store the contents, so must reallocate; if the new file is smaller than the old one, the space is enough but wasted, the test case 4 just tests the situation that the space is just full after the compaction, so not a single byte can be wasted. Therefore, in both cases we need to release the previous one and then apply for a new contiguous space, which is **dynamically allocation** rather than static allocation. **2.** The new contents occupies the same number of blocks as the previous(old) one. Great! Reuse the previous space to rewrite directly.

Detailed information about what to do when inequality holds(case 1 above): Firstly, be sure to release the old space in case there is not enough space left for the new files. The release operation only needs to erase the mark bit of the super block. Here, the release of the continuous mark bit is to use the bit operation(& the inverse code of MASK), and it doesn't matter whether the things on the storage are cleared or not. Then we call *get_empty_block* to get continuous *new_nblk* blocks. Actually, the super block is a bitmap. The strategy I use here is accessing with one *uint64_t*, which means accessing 64 bits data at a time. The below loop is to read the super block as an array of uint64_t and going through every value of uint64_t in it. If it is equal to UNIT64_MAX, representing all one, there must be no empty block, we should continue. Invoke *ffsl()* to get the subscript of the first empty bit and start the the probe with that subscript as the starting point.



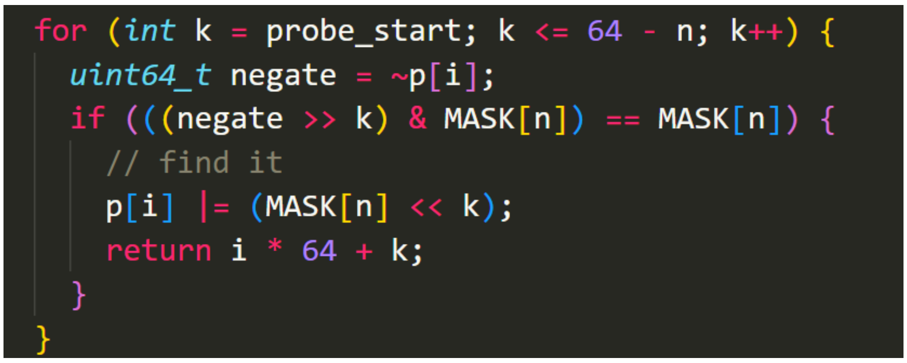
```
for (int i = 0; i < fs->SUPERBLOCK_SIZE / sizeof(uint64_t); i++)
  if (p[i] == UINT64_MAX)
    continue;
  int probe_start = ffsl(~p[i]);
```

Then there is a conditional branch, for example the first empty bit has the subscript 60, we want to allocate 10 consecutive blocks, since we only have 4 at last $(63 - 60 + 1)$, the only possible situation is to span two uint64_t if existence. That is to say there are six consecutive empty blocks at the start of the next uint64_t that makes it possible to get ten. Therefore, to do a judgement as shown below, *end_consecutive_nblk()* counts the longest consecutive empty blocks at the end, *start_consecutive_nblk()* counts the longest consecutive empty blocks at the beginning. If the total is greater than what we want – ten, that's it! We can invoke *get_block_from_2_unit()* to allocate across uint64_t.



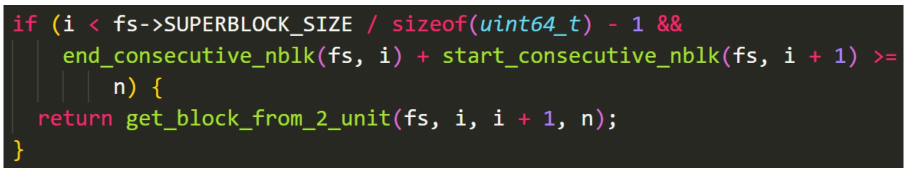
```
  if (probe_start + n - 1 >= 64 &&
      i < fs->SUPERBLOCK_SIZE / sizeof(uint64_t) - 1) {
    if (end_consecutive_nblk(fs, i) + start_consecutive_nblk(fs, i + 1) >=
        n) {
      return get_block_from_2_unit(fs, i, i + 1, n);
    }
```

If the first empty bit subscript is 1 and you want to allocate 10 consecutive blocks, since there are much more behind, this uint64_t is sufficient enough. As shown below, probe in itselft first.



```c
for (int k = probe_start; k <= 64 - n; k++) {
  uint64_t negate = ~p[i];
  if (((negate >> k) & MASK[n]) == MASK[n]) {
    // find it
    p[i] |= (MASK[n] << k);
    return i * 64 + k;
  }
}
```

If it probes in itself but doesn't find satisfied one, then to check whether it is possible to across uint64_t, as shown below. If still cannot find satisfied one, meaning that the fragment is so serious that no consecutive block can be use, return −1, to invoke **compact**.



```c
if (i < fs->SUPERBLOCK_SIZE / sizeof(uint64_t) - 1 &&
    end_consecutive_nblk(fs, i) + start_consecutive_nblk(fs, i + 1) >=
        n) {
  return get_block_from_2_unit(fs, i, i + 1, n);
}
```

Detailed information about **compact**. First traverse the super block in uint64_t, to check if there is any available block. If not, skip; if so, then pile the following blocks to the front. Get the length of the available blcok, and then calculate the length of the following consecutive non-empty blocks. After that, pile the whole of the following non-empty blocks in front of the available block length, then update the metadata information. Since the data in storage is piled, the address in FCB has to be updated, otherwise the data it point to will be wrong. For much more details, please check the source code.

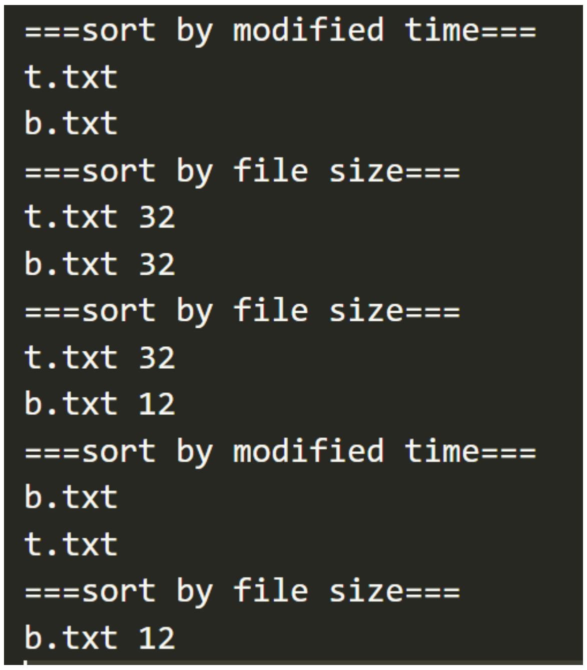
4 Problem Met in the Process

The first struggling problem is that I first implement it using static allocation. I think it is a very good implementation at first and passed all the test. Then, I have been told that it is not allowed! OK, actually it make sense because no real file system use static allocation. Then I watch the tut recording and check tut slides, I change it to dynamic allocation successfully. Another problem occur when I use recursion, "nvlink warning : Stack size for entry function

'_Z8mykernelPhs_' cannot be statically determined". I searched in Google and find some solutions in StackOverflow and CUDA official website, I only learn about that the GPU has a limited stack space and a limited number of registers per core causing the warning, but I still cannot solve it. In the end, I use loop instead of recursion to solve it, the code is not as elegant as before, so sad.

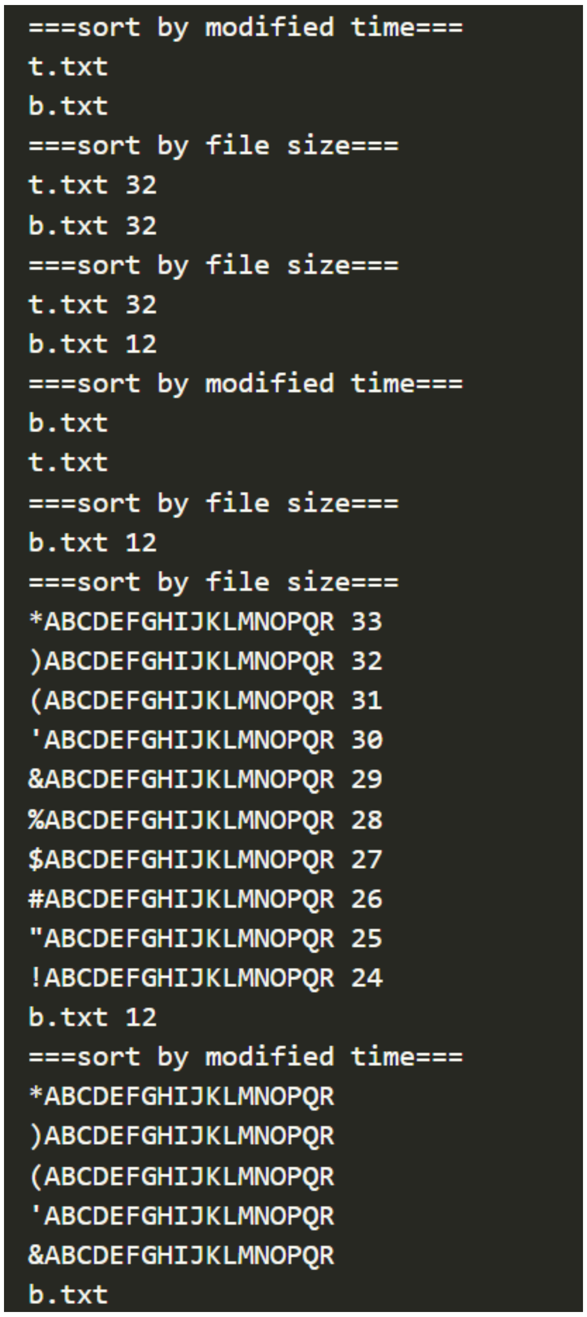
The last tricky problem I want to mention is in bonus part. The "soft" dictionary have four files "A.txt", "B.txt", "C.txt", "D.txt", and I output the size of the dictionary is 20, but the demo is 24. I am so confused and asked the USTF, he suggests me to read the homework description carefully, and then I found out that "\0" should be counted, invalid but valid! Great!

5 Screenshot of output



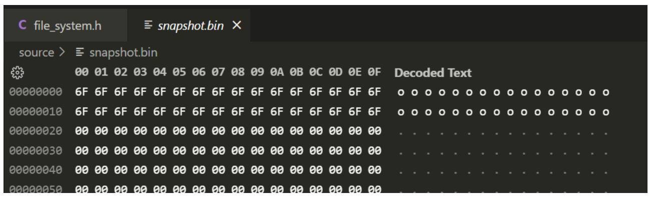
```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by file size===
t.txt 32
b.txt 12
===sort by modified time===
b.txt
t.txt
===sort by file size===
b.txt 12
```

Figure 2: TEST 1 Result



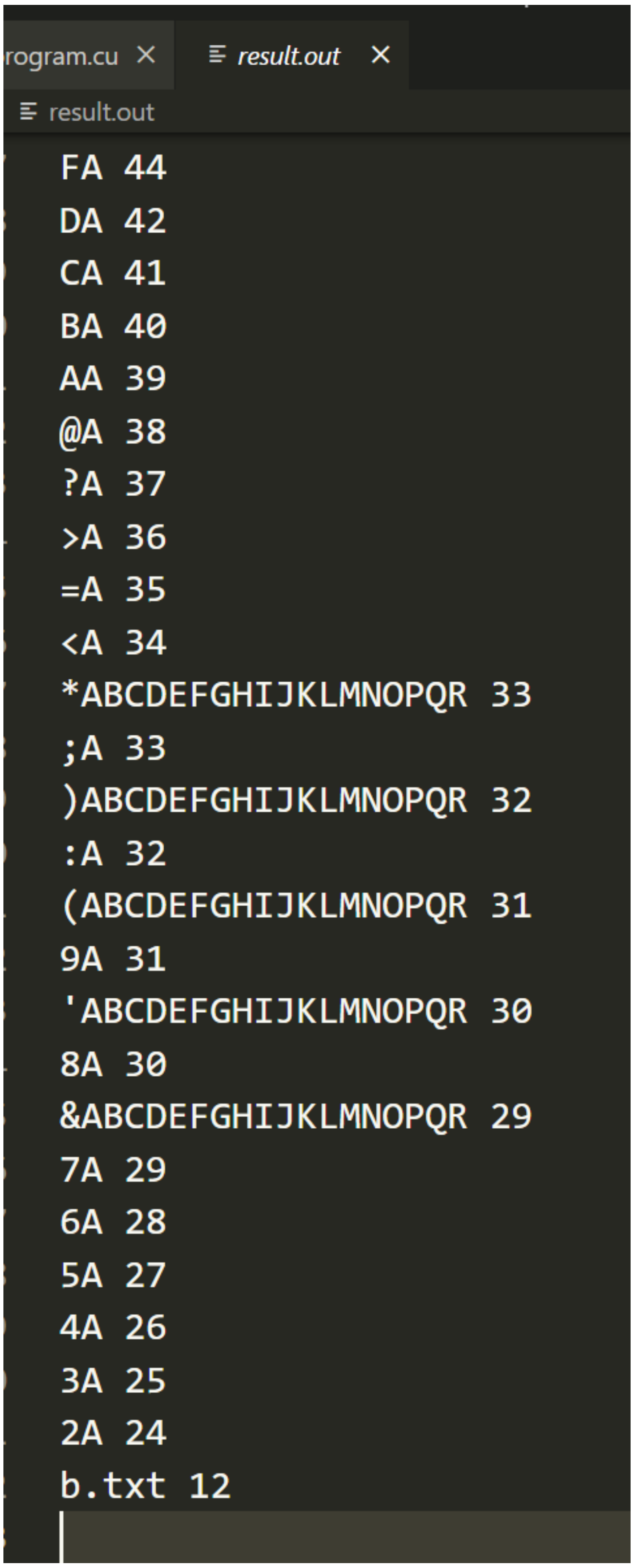
```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by file size===
t.txt 32
b.txt 12
===sort by modified time===
b.txt
t.txt
===sort by file size===
b.txt 12
===sort by file size===
*ABCDEFGHIJKLMNOPQR 33
)ABCDEFGHIJKLMNOPQR 32
(ABCDEFGHIJKLMNOPQR 31
'ABCDEFGHIJKLMNOPQR 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
```

Figure 3: TEST 2 Result



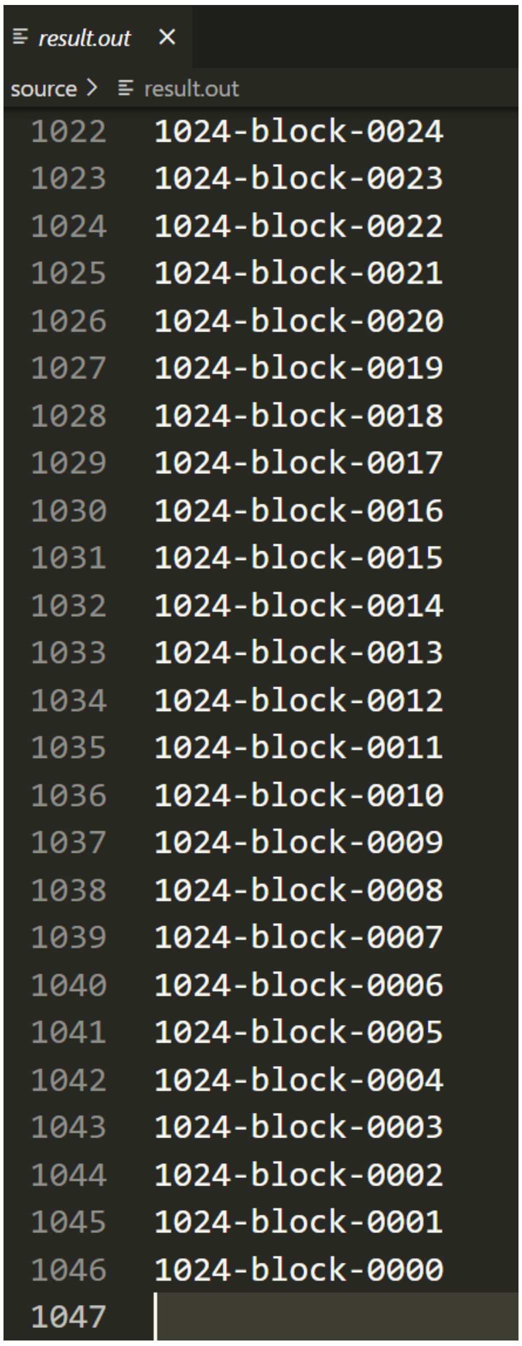
```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
00000000  6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F  o o o o o o o o o o o o o o o o
00000010  6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F 6F  o o o o o o o o o o o o o o o o
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
```

Figure 4: TEST 1&2 snapshot.bin



```
FA 44
DA 42
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
```

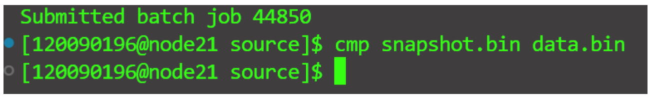
Figure 5: TEST 3 Result(part)



```
result.out
source > result.out
1024-block-0024
1024-block-0023
1024-block-0022
1024-block-0021
1024-block-0020
1024-block-0019
1024-block-0018
1024-block-0017
1024-block-0016
1024-block-0015
1024-block-0014
1024-block-0013
1024-block-0012
1024-block-0011
1024-block-0010
1024-block-0009
1024-block-0008
1024-block-0007
1024-block-0006
1024-block-0005
1024-block-0004
1024-block-0003
1024-block-0002
1024-block-0001
1024-block-0000
```

Figure 6: TEST 4 Result(part)

Please note that if there is no information after the command ”cmp file1 file2” indicates the contents of the two files are totally same

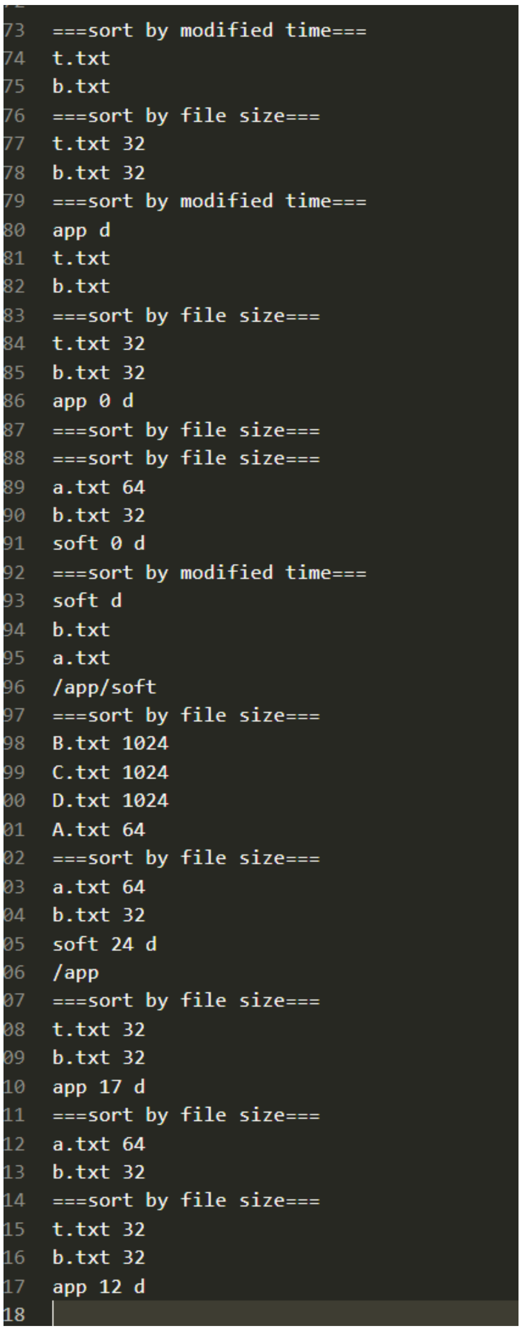


```
Submitted batch job 44850
[120090196@node21 source]$ cmp snapshot.bin data.bin
[120090196@node21 source]$
```

Figure 7: TEST 4 snapshot.bin

6 Bonus

Just do some modifications based on the basic part. Add a new struct, which is used to store the sub-files of the dictionary and the parent dictionary of that dictionary. Also, trace the global variable *cur_dir_fd* to locate which dictionary now is in. For more details, please check the source code. Below is the output for the given test case.



```
===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
```

Figure 8: Bonus

7 What I learn

1. Self study is important
2. Start as early as possible(?)
3. Before writing codes, we'd better understand what we need to do
4. Better understanding on File System and directory structure in the operating system
5. Define some helper functions is very helpful, the logic is much clearer than write the whole codes in one function

8 Attention

1. In **bonus** part, I add some **macros** in "**filesystem.h**".
2. In **source** part, I add "**#include stdio.h**" in "**filesystem.h**" because the fourth test case in "userprogram.cu" need to use "printf()" function
3. No other given template modified

9 In the end

Thanks for being patient to teach us and answer our questions in this term! Good Luck! We all have a bright future!