CSC3150 Assignment 4 Report

The bonus is partly implemented.

Part 0: Student Information

Student ID: 120090575

Student Name: 杜五洲 Du Wuzhou

Part 1: Assignment Overview

The assignment is mainly about simulating the file management system in the operating system. The source part simulates only the root directory, while the bonus part considers tree-structured directories. For the source part, **opening files, reading files and listing some file information** in the system are the main functions. The implementation mainly includes **compacting the files and contiguous allocation**. For the bonus part, the key point is still how to maintain a **compact structure of the file system**. Details are shown in the following part.

Part 2: Implementation Details

2.0 Design of System

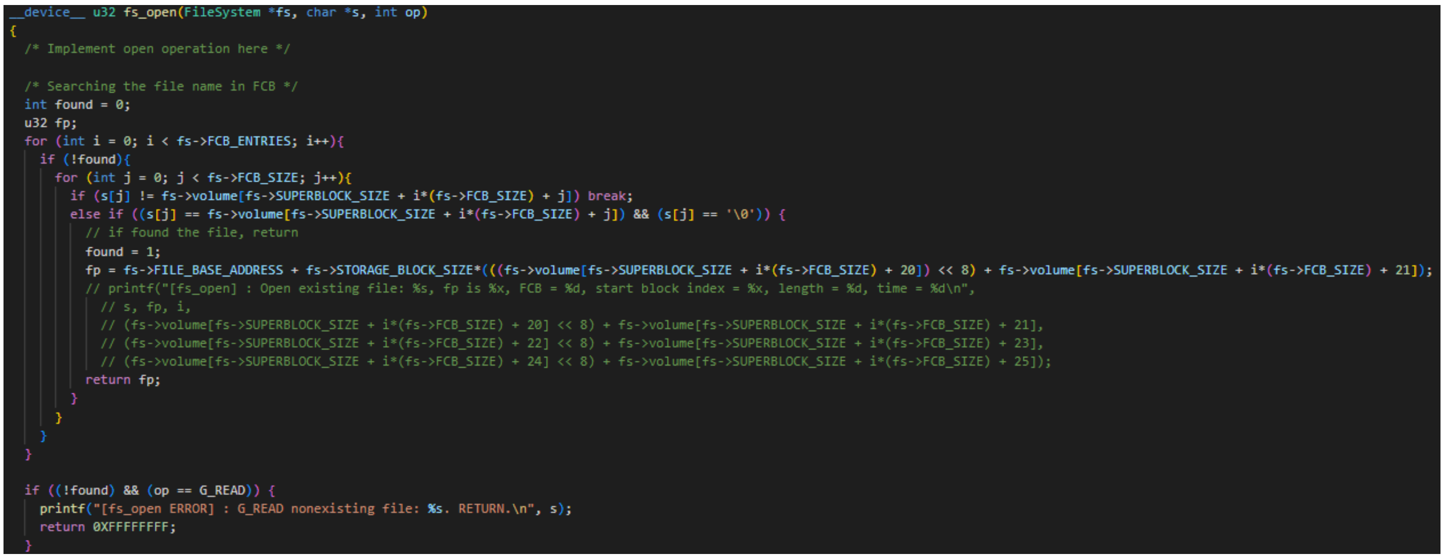
For the volume of the file system, the first 4KB is the **super block (Volume Control Block VCB),** in which every byte (8 bits) represents whether the 8 contiguous blocks have been allocated. If allocated, the corresponding bit is 1, otherwise 0. So this area is always interpreted with **binary representation**. The following 32KB is the **file control block (FCB). Each FCB is allocated with 32B, calculated by 32KB/1024(maximum number of files).** For every FCB, the structure to store the **meta information** of the corresponding file is shown as follows:

File name (20B)	Start block index (2B)	Length of file (2B)	Modified time (2B)	Created time (2B)

While in the bonus part, FCB should store more meta information due to the tree-structured directory, where the **identity of this file** is **a directory (1) or a file (0)**, represented in the **29th byte (though it wastes the other 31 bits)**. The parent directory index of the **root directory is 0xffff**. The depth of the file or directory is the distance between itself and the root directory + 1. For example, the **root directory** has **depth 1**.

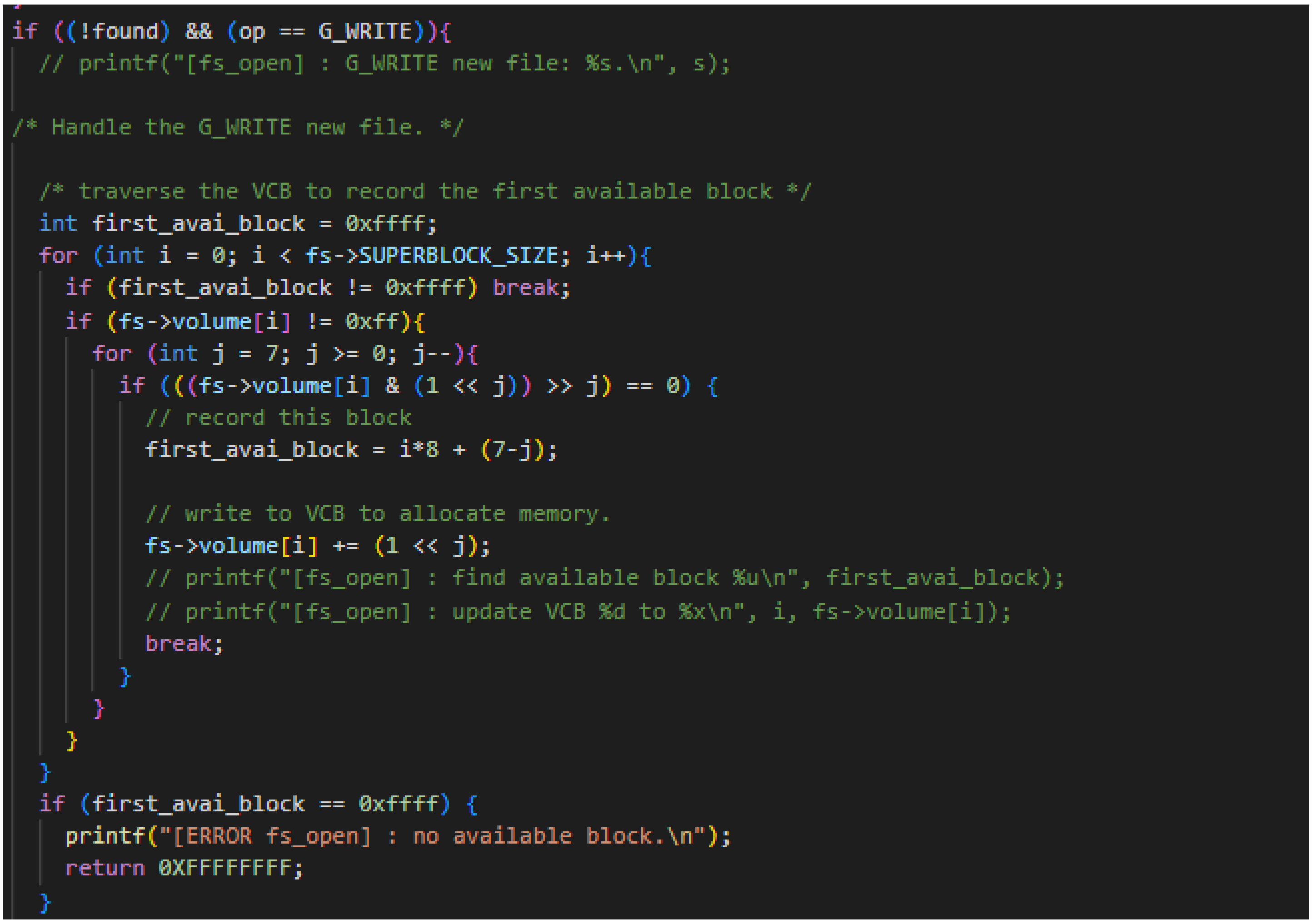
File name (20B)	Start block index (2B)	Length of file (2B)	Modified time (2B)	Created time (2B)

2.1 fs_open

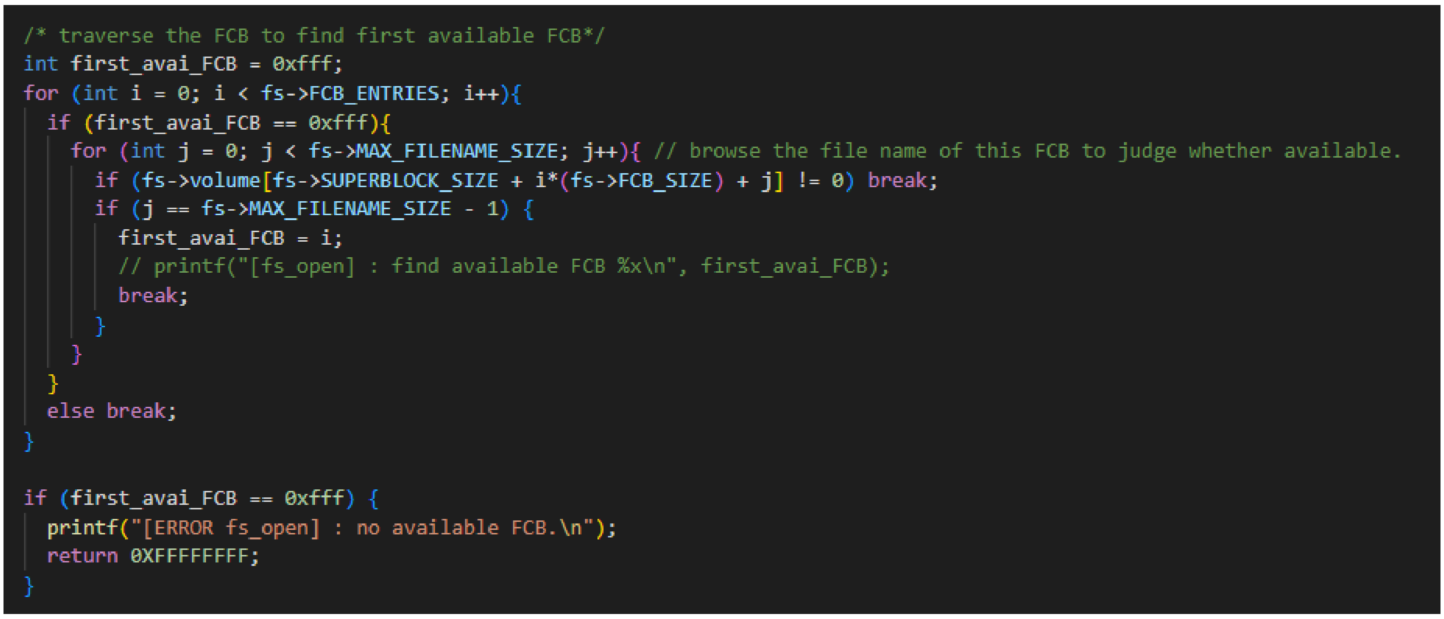


```c
__device__ u32 fs_open(FileSystem *fs, char *s, int op)
{
  /* Implement open operation here */

  /* Searching the file name in FCB */
  int found = 0;
  u32 fp;
  for (int i = 0; i < fs->FCB_ENTRIES; i++){
    if (!found){
      for (int j = 0; j < fs->FCB_SIZE; j++){
        if (s[j] != fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + j]) break;
        else if ((s[j] == fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + j]) && (s[j] == '\0')) {
          // if found the file, return
          found = 1;
          fp = fs->FILE_BASE_ADDRESS + fs->STORAGE_BLOCK_SIZE*(((fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 20]) << 8) + fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 21]);
          // printf("[fs_open] : Open existing file: %s, fp is %x, FCB = %d, start block index = %x, length = %d, time = %d\n",
            // s, fp, i,
            // (fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 20] << 8) + fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 21],
            // (fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 22] << 8) + fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 23],
            // (fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 24] << 8) + fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 25]);
          return fp;
        }
      }
    }
  }

  if ((!found) && (op == G_READ)) {
    printf("[fs_open ERROR] : G_READ nonexisting file: %s. RETURN.\n", s);
    return 0XFFFFFFFF;
  }
```

`fs_open` function returns a **pointer to the physical address** of the wanted file. Naturally, first consider whether this file is in the FCB. So try to search for its name in FCB. If the file already exists in FCB, then return its starting address with the help of metadata stored in FCB.

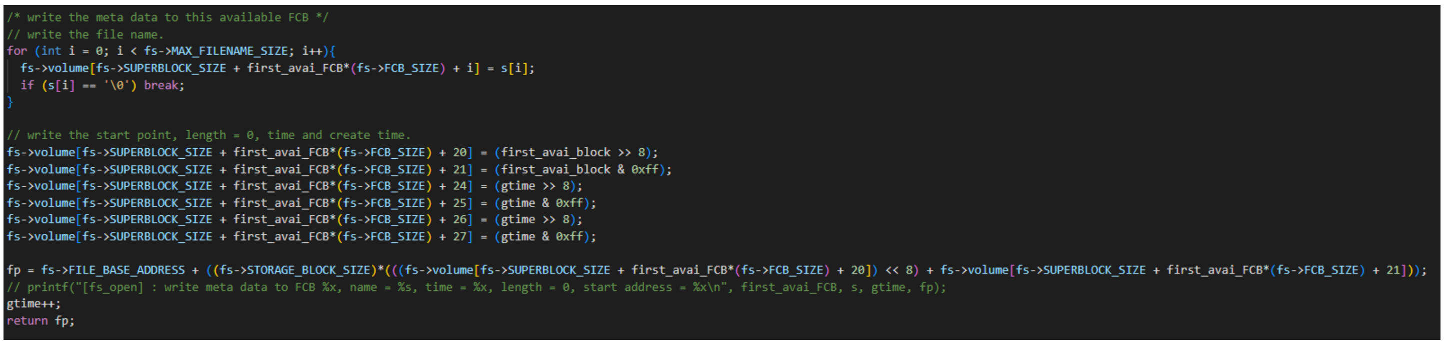


```c
if ((!found) && (op == G_WRITE)){
  // printf("[fs_open] : G_WRITE new file: %s.\n", s);

/* Handle the G_WRITE new file. */

  /* traverse the VCB to record the first available block */
  int first_avai_block = 0xffff;
  for (int i = 0; i < fs->SUPERBLOCK_SIZE; i++){
    if (first_avai_block != 0xffff) break;
    if (fs->volume[i] != 0xff){
      for (int j = 7; j >= 0; j--){
        if (((fs->volume[i] & (1 << j)) >> j) == 0) {
          // record this block
          first_avai_block = i*8 + (7-j);

          // write to VCB to allocate memory.
          fs->volume[i] += (1 << j);
          // printf("[fs_open] : find available block %u\n", first_avai_block);
          // printf("[fs_open] : update VCB %d to %x\n", i, fs->volume[i]);
          break;
        }
      }
    }
  }
  if (first_avai_block == 0xffff) {
    printf("[ERROR fs_open] : no available block.\n");
    return 0XFFFFFFFF;
  }
```



```c
/* traverse the FCB to find first available FCB*/
int first_avai_FCB = 0xfff;
for (int i = 0; i < fs->FCB_ENTRIES; i++){
  if (first_avai_FCB == 0xfff){
    for (int j = 0; j < fs->MAX_FILENAME_SIZE; j++){ // browse the file name of this FCB to judge whether available.
      if (fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + j] != 0) break;
      if (j == fs->MAX_FILENAME_SIZE - 1) {
        first_avai_FCB = i;
        // printf("[fs_open] : find available FCB %x\n", first_avai_FCB);
        break;
      }
    }
  }
  else break;
}

if (first_avai_FCB == 0xfff) {
  printf("[ERROR fs_open] : no available FCB.\n");
  return 0XFFFFFFFF;
}
```

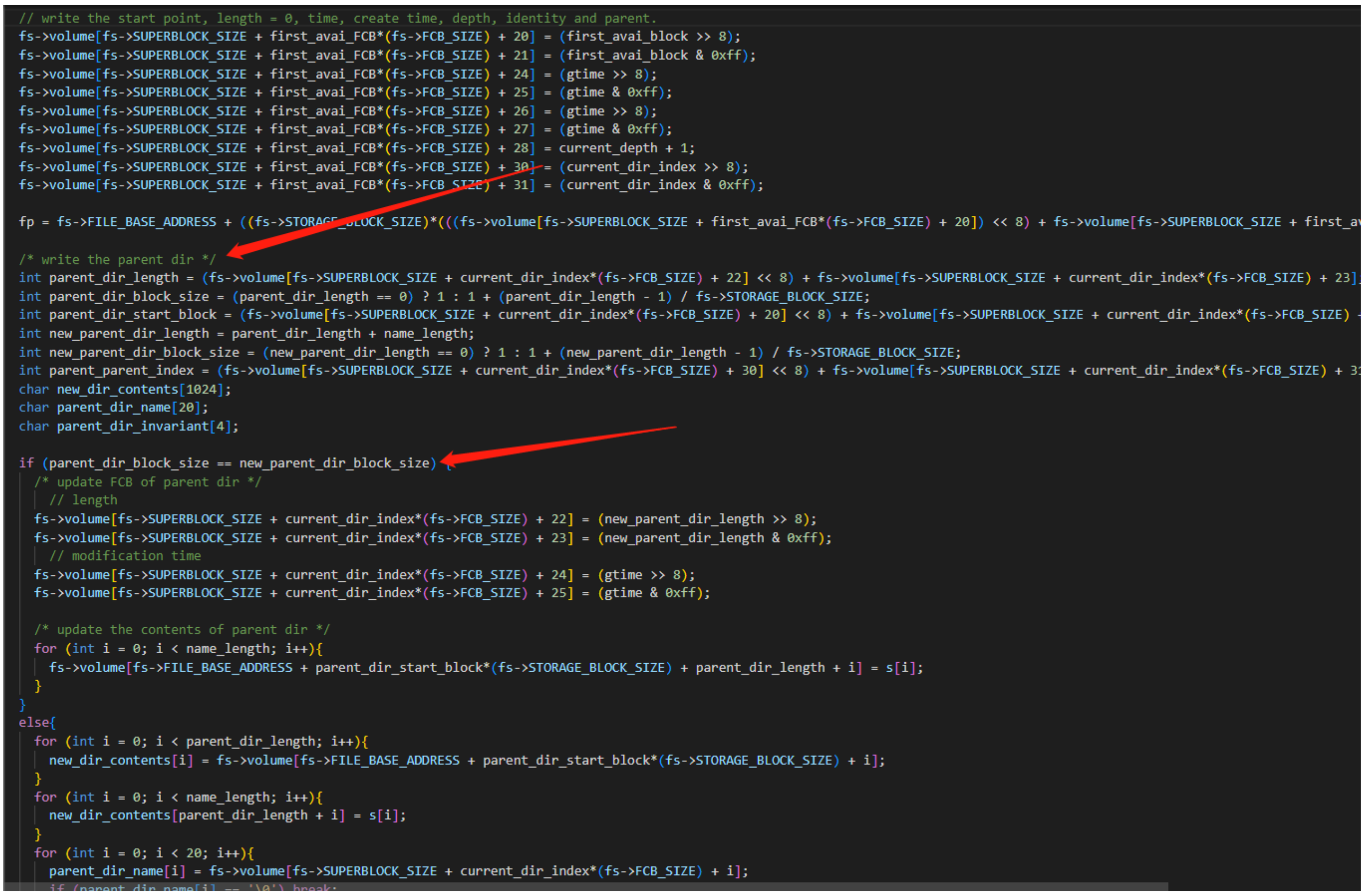
If not found, then consider the **G_WRITE** mode. Now the program tries to find the first available block and available FCB to store the newly created file. To find an available place, just search the FCB and VCB from head to tail. During the process of searching for VCB, also update the first available VCB to allocate a new block for the newly created file.



```c
/* write the meta data to this available FCB */
// write the file name.
for (int i = 0; i < fs->MAX_FILENAME_SIZE; i++){
  fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + i] = s[i];
  if (s[i] == '\0') break;
}

// write the start point, length = 0, time and create time.
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 20] = (first_avai_block >> 8);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 21] = (first_avai_block & 0xff);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 24] = (gtime >> 8);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 25] = (gtime & 0xff);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 26] = (gtime >> 8);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 27] = (gtime & 0xff);

fp = fs->FILE_BASE_ADDRESS + ((fs->STORAGE_BLOCK_SIZE)*(((fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 20]) << 8) + fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 21]));
// printf("[fs_open] : write meta data to FCB %x, name = %s, time = %x, length = 0, start address = %x\n", first_avai_FCB, s, gtime, fp);
gtime++;
return fp;
```

After finding the available block and FCB, store the metadata of the file in FCB. Since it's a vacant file, there is no need to update the file content.

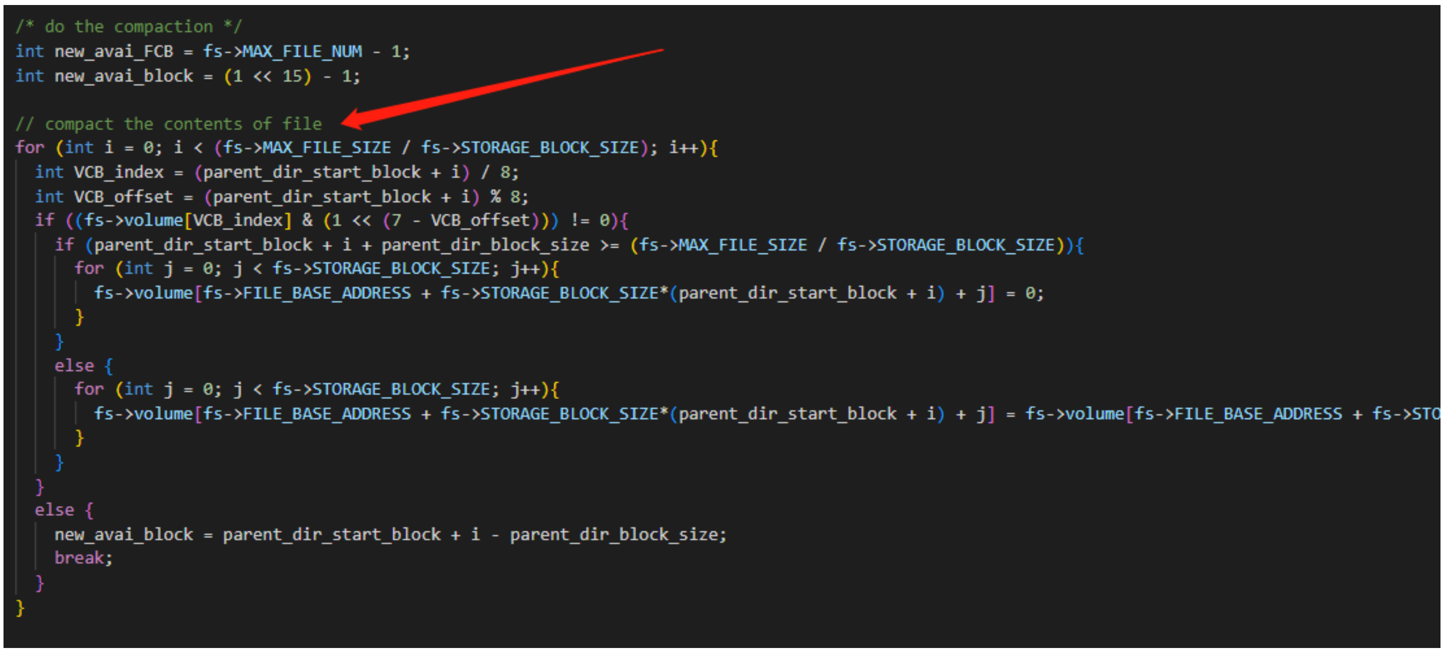
By now, `fs_open` in the source part ends. However, in the **bonus** part, not only considering whether the file name matches, whether the filename is in the same directory and whether the file number limit is reached also matter. Only when the matched name file is in the current directory is the file considered found. What's more, the meta data of this newly created file is more than the source part. But the extras in the bonus part mentioned above are trivial. **More importantly**, after opening this newly created file, the **parent directory data should be updated**. Since this is a new file, the **file content** of the directory is added with the name of the new file. And the metadata **'length' and 'modified time'** should also be updated. ***But take care, every time the length changes, consider whether the block size of the file changes. If the block size changes, compaction is needed both in the bonus part and source part. And the flow of compaction is the same.***



```c
// write the start point, length = 0, time, create time, depth, identity and parent.
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 20] = (first_avai_block >> 8);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 21] = (first_avai_block & 0xff);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 24] = (gtime >> 8);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 25] = (gtime & 0xff);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 26] = (gtime >> 8);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 27] = (gtime & 0xff);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 28] = current_depth + 1;
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 30] = (current_dir_index >> 8);
fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 31] = (current_dir_index & 0xff);

fp = fs->FILE_BASE_ADDRESS + ((fs->STORAGE_BLOCK_SIZE)*(((fs->volume[fs->SUPERBLOCK_SIZE + first_avai_FCB*(fs->FCB_SIZE) + 20]) << 8) + fs->volume[fs->SUPERBLOCK_SIZE + first_a

/* write the parent dir */
int parent_dir_length = (fs->volume[fs->SUPERBLOCK_SIZE + current_dir_index*(fs->FCB_SIZE) + 22] << 8) + fs->volume[fs->SUPERBLOCK_SIZE + current_dir_index*(fs->FCB_SIZE) + 23]
int parent_dir_block_size = (parent_dir_length == 0) ? 1 : 1 + (parent_dir_length - 1) / fs->STORAGE_BLOCK_SIZE;
int parent_dir_start_block = (fs->volume[fs->SUPERBLOCK_SIZE + current_dir_index*(fs->FCB_SIZE) + 20] << 8) + fs->volume[fs->SUPERBLOCK_SIZE + current_dir_index*(fs->FCB_SIZE) 
int new_parent_dir_length = parent_dir_length + name_length;
int new_parent_dir_block_size = (new_parent_dir_length == 0) ? 1 : 1 + (new_parent_dir_length - 1) / fs->STORAGE_BLOCK_SIZE;
int parent_parent_index = (fs->volume[fs->SUPERBLOCK_SIZE + current_dir_index*(fs->FCB_SIZE) + 30] << 8) + fs->volume[fs->SUPERBLOCK_SIZE + current_dir_index*(fs->FCB_SIZE) + 3
char new_dir_contents[1024];
char parent_dir_name[20];
char parent_dir_invariant[4];

if (parent_dir_block_size == new_parent_dir_block_size) {
  /* update FCB of parent dir */
    // length
  fs->volume[fs->SUPERBLOCK_SIZE + current_dir_index*(fs->FCB_SIZE) + 22] = (new_parent_dir_length >> 8);
  fs->volume[fs->SUPERBLOCK_SIZE + current_dir_index*(fs->FCB_SIZE) + 23] = (new_parent_dir_length & 0xff);
    // modification time
  fs->volume[fs->SUPERBLOCK_SIZE + current_dir_index*(fs->FCB_SIZE) + 24] = (gtime >> 8);
  fs->volume[fs->SUPERBLOCK_SIZE + current_dir_index*(fs->FCB_SIZE) + 25] = (gtime & 0xff);

  /* update the contents of parent dir */
  for (int i = 0; i < name_length; i++){
    fs->volume[fs->FILE_BASE_ADDRESS + parent_dir_start_block*(fs->STORAGE_BLOCK_SIZE) + parent_dir_length + i] = s[i];
  }
}
else{
  for (int i = 0; i < parent_dir_length; i++){
    new_dir_contents[i] = fs->volume[fs->FILE_BASE_ADDRESS + parent_dir_start_block*(fs->STORAGE_BLOCK_SIZE) + i];
  }
  for (int i = 0; i < name_length; i++){
    new_dir_contents[parent_dir_length + i] = s[i];
  }
  for (int i = 0; i < 20; i++){
    parent_dir_name[i] = fs->volume[fs->SUPERBLOCK_SIZE + current_dir_index*(fs->FCB_SIZE) + i];
    if (parent_dir_name[i] == '\0') break;
```

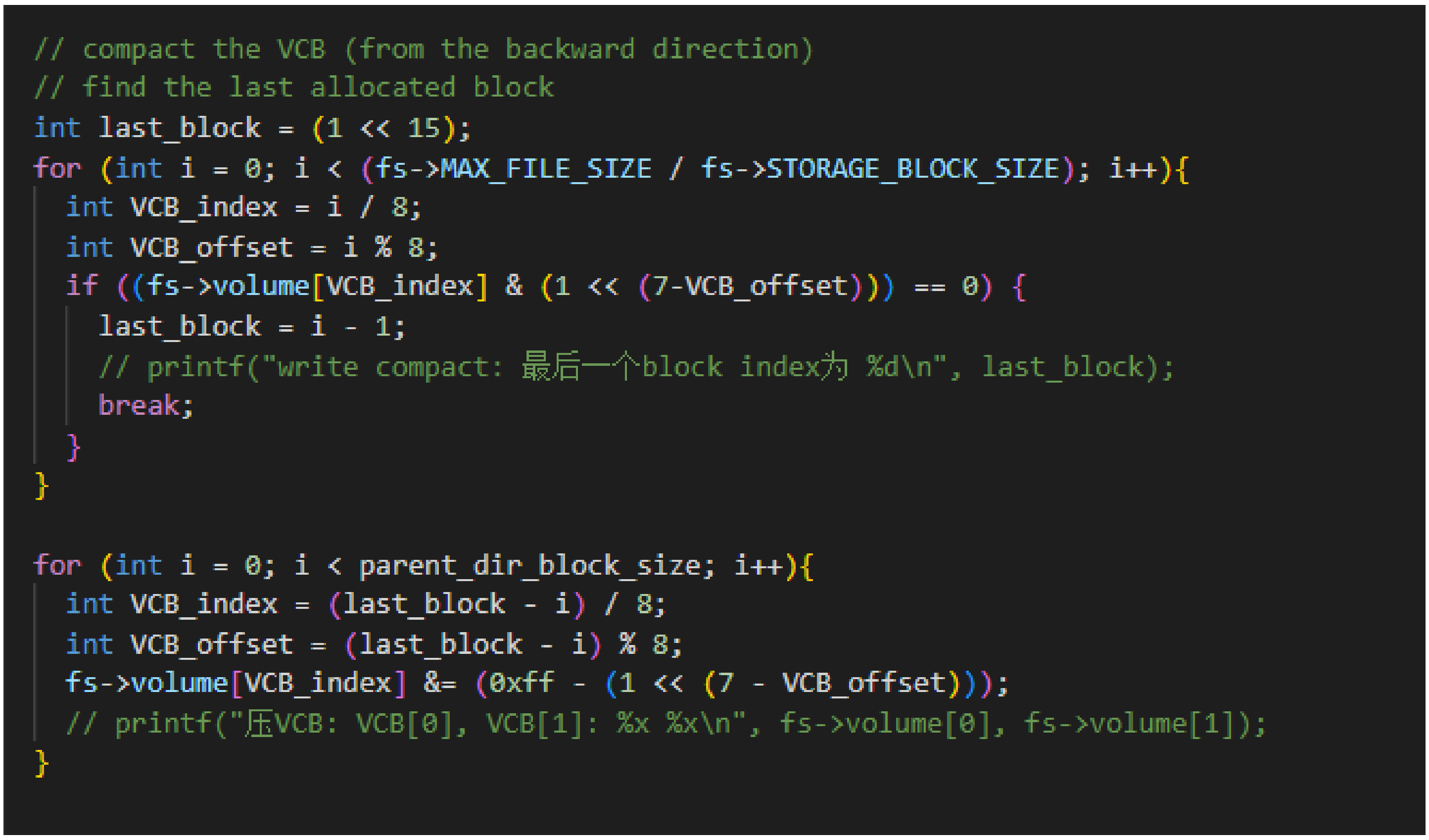
The process of handling the length change is as follows:

1. **investigating whether the block size changes ->**
2. **if no change or this is the last file in the FCB, thank god, no need to compact ->**
3. **else remove this changed size file and do the compaction (compact the file contents, compact the VCB and update the FCB information)->**
4. **add this changed size file to the end of the block and FCB**



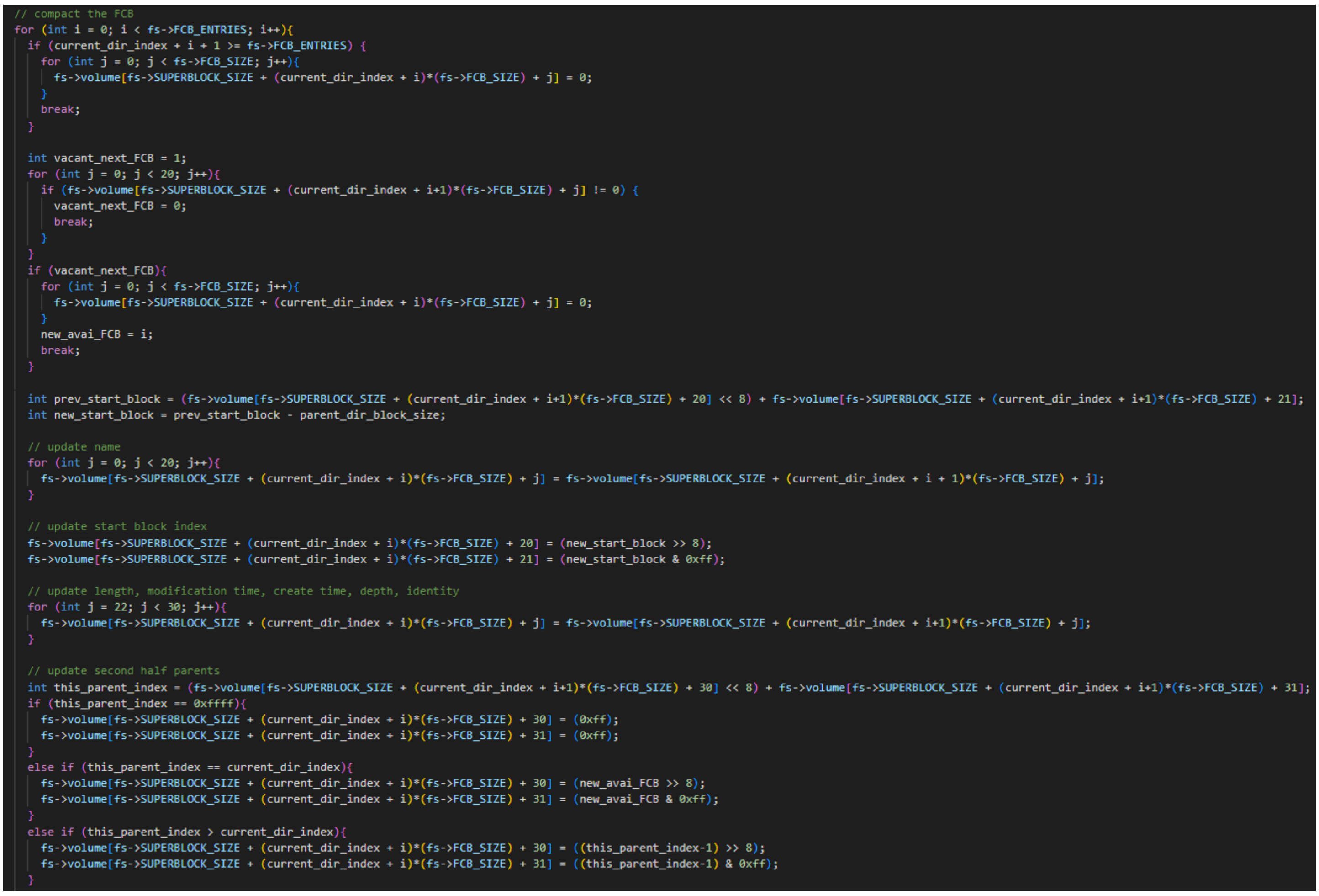
```c
/* do the compaction */
int new_avai_FCB = fs->MAX_FILE_NUM - 1;
int new_avai_block = (1 << 15) - 1;

// compact the contents of file
for (int i = 0; i < (fs->MAX_FILE_SIZE / fs->STORAGE_BLOCK_SIZE); i++){
  int VCB_index = (parent_dir_start_block + i) / 8;
  int VCB_offset = (parent_dir_start_block + i) % 8;
  if ((fs->volume[VCB_index] & (1 << (7 - VCB_offset))) != 0){
    if (parent_dir_start_block + i + parent_dir_block_size >= (fs->MAX_FILE_SIZE / fs->STORAGE_BLOCK_SIZE)){
      for (int j = 0; j < fs->STORAGE_BLOCK_SIZE; j++){
        fs->volume[fs->FILE_BASE_ADDRESS + fs->STORAGE_BLOCK_SIZE*(parent_dir_start_block + i) + j] = 0;
      }
    }
    else {
      for (int j = 0; j < fs->STORAGE_BLOCK_SIZE; j++){
        fs->volume[fs->FILE_BASE_ADDRESS + fs->STORAGE_BLOCK_SIZE*(parent_dir_start_block + i) + j] = fs->volume[fs->FILE_BASE_ADDRESS + fs->STO
      }
    }
  }
  else {
    new_avai_block = parent_dir_start_block + i - parent_dir_block_size;
    break;
  }
}
```

The key point of the implementation is **how to compact**. First, compact the file contents block by block from the starting block, which is often the starting block of the removed file. Rewrite the block contents. It is intuitional that the **new contents are just from the block after an offset (equal to the removed block size)**.

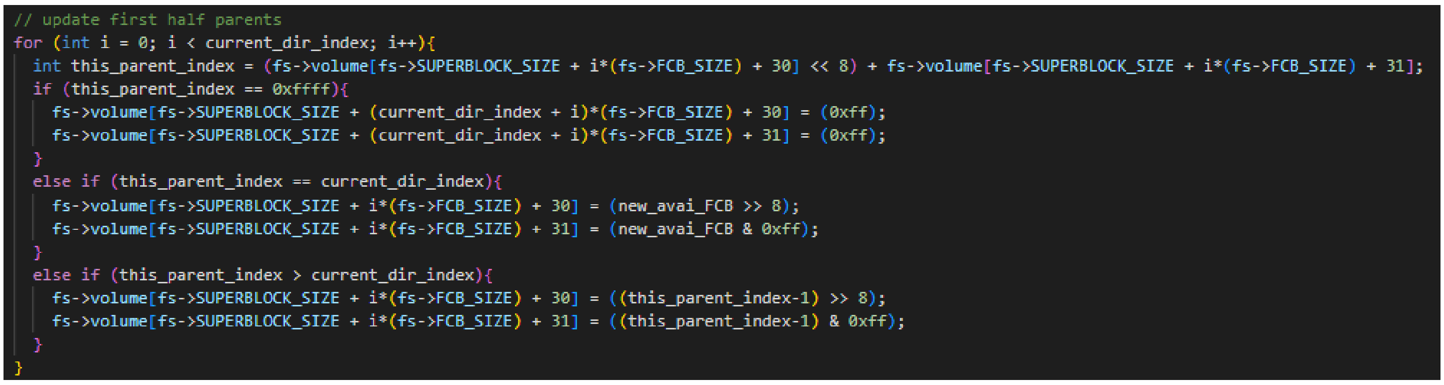


```
// compact the VCB (from the backward direction)
// find the last allocated block
int last_block = (1 << 15);
for (int i = 0; i < (fs->MAX_FILE_SIZE / fs->STORAGE_BLOCK_SIZE); i++){
  int VCB_index = i / 8;
  int VCB_offset = i % 8;
  if ((fs->volume[VCB_index] & (1 << (7-VCB_offset))) == 0) {
    last_block = i - 1;
    // printf("write compact: 最后一个block index为 %d\n", last_block);
    break;
  }
}

for (int i = 0; i < parent_dir_block_size; i++){
  int VCB_index = (last_block - i) / 8;
  int VCB_offset = (last_block - i) % 8;
  fs->volume[VCB_index] &= (0xff - (1 << (7 - VCB_offset)));
  // printf("压VCB: VCB[0], VCB[1]: %x %x\n", fs->volume[0], fs->volume[1]);
}
```

Secondly, compact the VCB. The VCB is just all 1 in the previous state, so find the last VCB, which in the byte representation is not 0xff. Then, from the last VCB to compact backward, the **last few bits (the number of bits equal to the removed block size) should be updated from 1 to 0.**

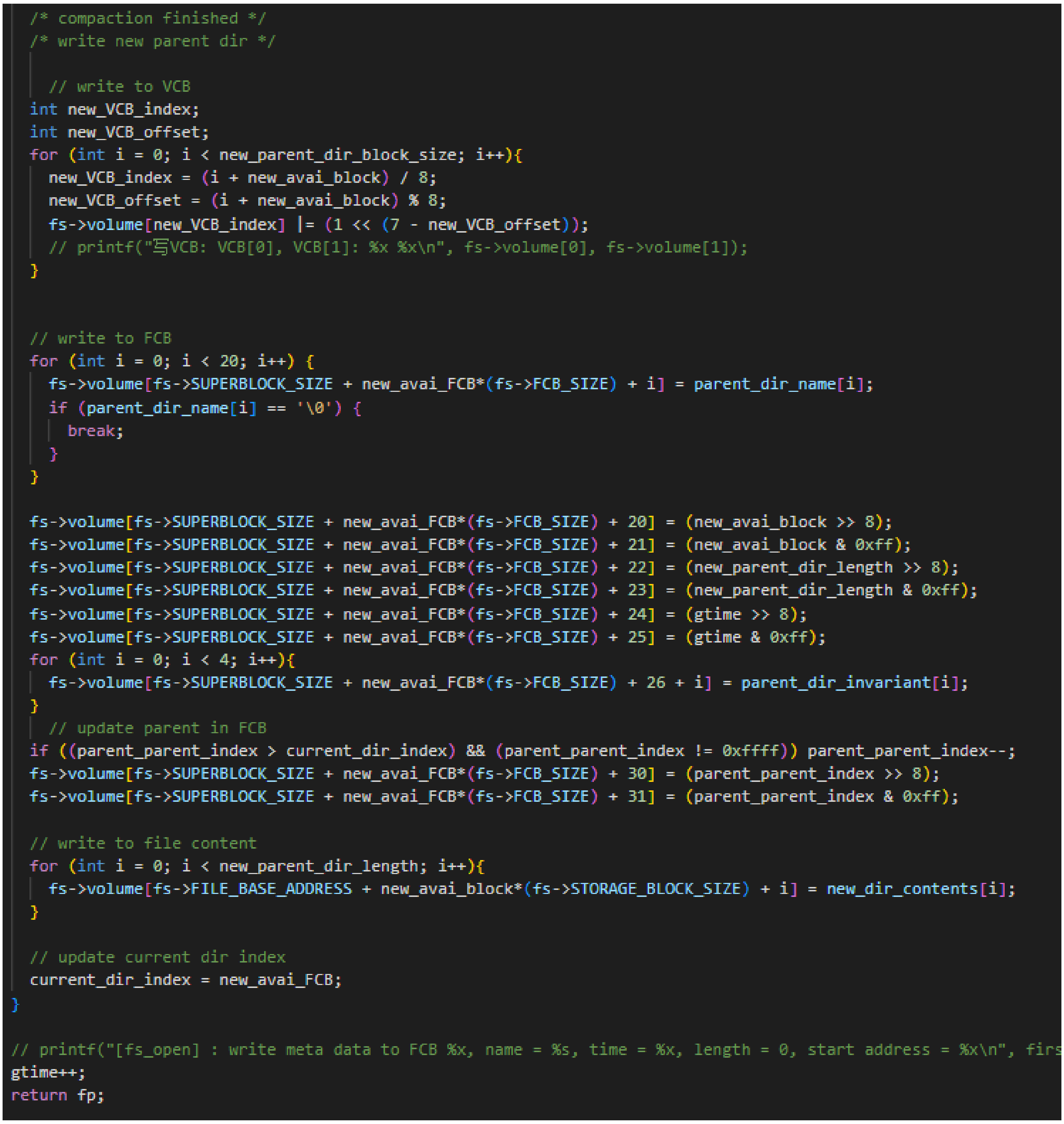


```c
// compact the FCB
for (int i = 0; i < fs->FCB_ENTRIES; i++){
  if (current_dir_index + i + 1 >= fs->FCB_ENTRIES) {
    for (int j = 0; j < fs->FCB_SIZE; j++){
      fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + j] = 0;
    }
    break;
  }

  int vacant_next_FCB = 1;
  for (int j = 0; j < 20; j++){
    if (fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i+1)*(fs->FCB_SIZE) + j] != 0) {
      vacant_next_FCB = 0;
      break;
    }
  }
  if (vacant_next_FCB){
    for (int j = 0; j < fs->FCB_SIZE; j++){
      fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + j] = 0;
    }
    new_avai_FCB = i;
    break;
  }

  int prev_start_block = (fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i+1)*(fs->FCB_SIZE) + 20] << 8) + fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i+1)*(fs->FCB_SIZE) + 21];
  int new_start_block = prev_start_block - parent_dir_block_size;

  // update name
  for (int j = 0; j < 20; j++){
    fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + j] = fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i + 1)*(fs->FCB_SIZE) + j];
  }

  // update start block index
  fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + 20] = (new_start_block >> 8);
  fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + 21] = (new_start_block & 0xff);

  // update length, modification time, create time, depth, identity
  for (int j = 22; j < 30; j++){
    fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + j] = fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i+1)*(fs->FCB_SIZE) + j];
  }

  // update second half parents
  int this_parent_index = (fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i+1)*(fs->FCB_SIZE) + 30] << 8) + fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i+1)*(fs->FCB_SIZE) + 31];
  if (this_parent_index == 0xffff){
    fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + 30] = (0xff);
    fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + 31] = (0xff);
  }
  else if (this_parent_index == current_dir_index){
    fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + 30] = (new_avai_FCB >> 8);
    fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + 31] = (new_avai_FCB & 0xff);
  }
  else if (this_parent_index > current_dir_index){
    fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + 30] = ((this_parent_index-1) >> 8);
    fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + 31] = ((this_parent_index-1) & 0xff);
  }
```



```c
// update first half parents
for (int i = 0; i < current_dir_index; i++){
  int this_parent_index = (fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 30] << 8) + fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 31];
  if (this_parent_index == 0xffff){
    fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + 30] = (0xff);
    fs->volume[fs->SUPERBLOCK_SIZE + (current_dir_index + i)*(fs->FCB_SIZE) + 31] = (0xff);
  }
  else if (this_parent_index == current_dir_index){
    fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 30] = (new_avai_FCB >> 8);
    fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 31] = (new_avai_FCB & 0xff);
  }
  else if (this_parent_index > current_dir_index){
    fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 30] = ((this_parent_index-1) >> 8);
    fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + 31] = ((this_parent_index-1) & 0xff);
  }
}
```

Finally, update FCB. The FCB contains the information mentioned in the design part. However, one more thing needs to be consisdered is that, **in bonus part**, the **parent directory index** is stored in every FCB. But think about the process of compaction, **some FCB are moved forward 1 unit**! And when **the removed file is a directory, its index is newly allocated**! So when updating the FCB, the parent directory index should be handled carefully. And a trivial case is that the **root directory's parent index should remain the same**.



```
    /* compaction finished */
    /* write new parent dir */

        // write to VCB
    int new_VCB_index;
    int new_VCB_offset;
    for (int i = 0; i < new_parent_dir_block_size; i++){
        new_VCB_index = (i + new_avai_block) / 8;
        new_VCB_offset = (i + new_avai_block) % 8;
        fs->volume[new_VCB_index] |= (1 << (7 - new_VCB_offset));
        // printf("写VCB: VCB[0], VCB[1]: %x %x\n", fs->volume[0], fs->volume[1]);
    }


    // write to FCB
    for (int i = 0; i < 20; i++) {
        fs->volume[fs->SUPERBLOCK_SIZE + new_avai_FCB*(fs->FCB_SIZE) + i] = parent_dir_name[i];
        if (parent_dir_name[i] == '\0') {
            break;
        }
    }

    fs->volume[fs->SUPERBLOCK_SIZE + new_avai_FCB*(fs->FCB_SIZE) + 20] = (new_avai_block >> 8);
    fs->volume[fs->SUPERBLOCK_SIZE + new_avai_FCB*(fs->FCB_SIZE) + 21] = (new_avai_block & 0xff);
    fs->volume[fs->SUPERBLOCK_SIZE + new_avai_FCB*(fs->FCB_SIZE) + 22] = (new_parent_dir_length >> 8);
    fs->volume[fs->SUPERBLOCK_SIZE + new_avai_FCB*(fs->FCB_SIZE) + 23] = (new_parent_dir_length & 0xff);
    fs->volume[fs->SUPERBLOCK_SIZE + new_avai_FCB*(fs->FCB_SIZE) + 24] = (gtime >> 8);
    fs->volume[fs->SUPERBLOCK_SIZE + new_avai_FCB*(fs->FCB_SIZE) + 25] = (gtime & 0xff);
    for (int i = 0; i < 4; i++){
        fs->volume[fs->SUPERBLOCK_SIZE + new_avai_FCB*(fs->FCB_SIZE) + 26 + i] = parent_dir_invariant[i];
    }
        // update parent in FCB
    if ((parent_parent_index > current_dir_index) && (parent_parent_index != 0xffff)) parent_parent_index--;
    fs->volume[fs->SUPERBLOCK_SIZE + new_avai_FCB*(fs->FCB_SIZE) + 30] = (parent_parent_index >> 8);
    fs->volume[fs->SUPERBLOCK_SIZE + new_avai_FCB*(fs->FCB_SIZE) + 31] = (parent_parent_index & 0xff);

    // write to file content
    for (int i = 0; i < new_parent_dir_length; i++){
        fs->volume[fs->FILE_BASE_ADDRESS + new_avai_block*(fs->STORAGE_BLOCK_SIZE) + i] = new_dir_contents[i];
    }

    // update current dir index
    current_dir_index = new_avai_FCB;
}

// printf("[fs_open] : write meta data to FCB %x, name = %s, time = %x, length = 0, start address = %x\n", firs
gtime++;
return fp;
```

After compaction, add the removed file to the end of other files. **This is generally the whole process and the important things of compaction. The following statement will not repeat it.**

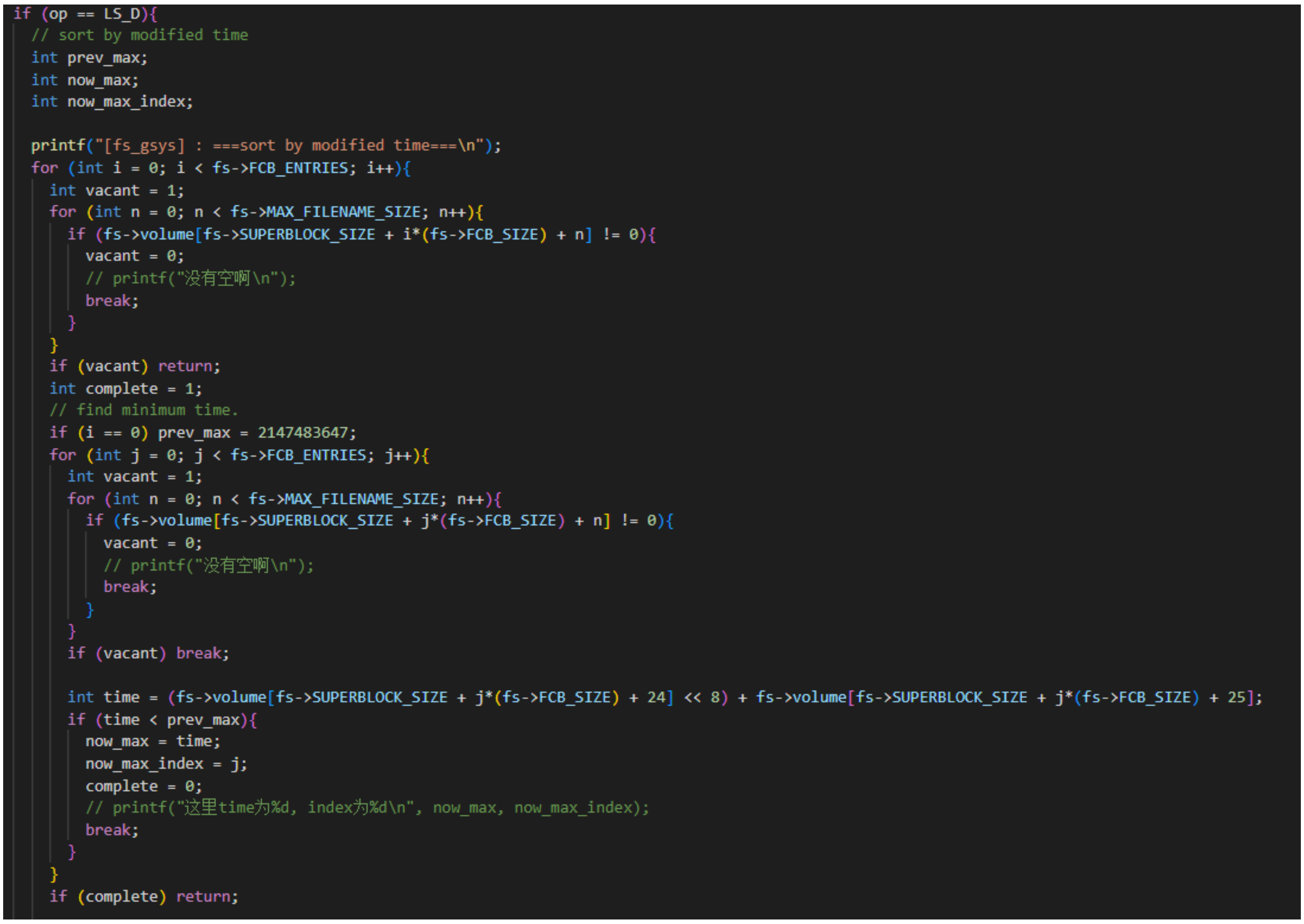
2.2 fs_write

Generally, the writing process has a similar flow to the opening process (fetech the file in FCB -> check if there is any block size change -> handle it). But since the opening process has created the file already, the writing process should **definitely verify the file to write, otherwise raise an error.**

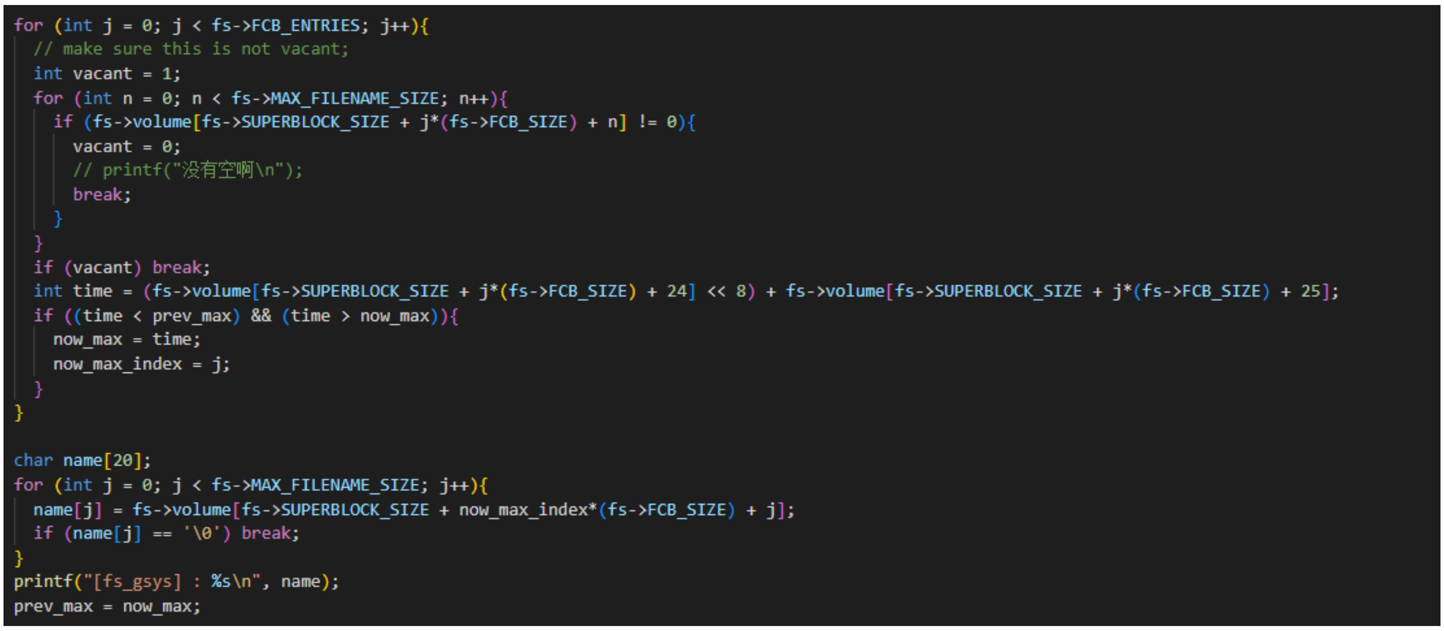
2.3 fs_gsys

2.3.1 LS_D and LS_S

The main idea is to search in FCB iteratively. In every iteration, **find the most fit file** to print out (how to define the 'most fit' depends on **file's modified time, size, created time, LS_S and LS_D**. But it's trivial here.) To filter and find the candidate to print out, **maintaining the most fit file information from the last iteration** is needed.



```c
if (op == LS_D){
  // sort by modified time
  int prev_max;
  int now_max;
  int now_max_index;

  printf("[fs_gsys] : ===sort by modified time===\n");
  for (int i = 0; i < fs->FCB_ENTRIES; i++){
    int vacant = 1;
    for (int n = 0; n < fs->MAX_FILENAME_SIZE; n++){
      if (fs->volume[fs->SUPERBLOCK_SIZE + i*(fs->FCB_SIZE) + n] != 0){
        vacant = 0;
        // printf("没有空啊\n");
        break;
      }
    }
    if (vacant) return;
    int complete = 1;
    // find minimum time.
    if (i == 0) prev_max = 2147483647;
    for (int j = 0; j < fs->FCB_ENTRIES; j++){
      int vacant = 1;
      for (int n = 0; n < fs->MAX_FILENAME_SIZE; n++){
        if (fs->volume[fs->SUPERBLOCK_SIZE + j*(fs->FCB_SIZE) + n] != 0){
          vacant = 0;
          // printf("没有空啊\n");
          break;
        }
      }
      if (vacant) break;

      int time = (fs->volume[fs->SUPERBLOCK_SIZE + j*(fs->FCB_SIZE) + 24] << 8) + fs->volume[fs->SUPERBLOCK_SIZE + j*(fs->FCB_SIZE) + 25];
      if (time < prev_max){
        now_max = time;
        now_max_index = j;
        complete = 0;
        // printf("这里time为%d, index为%d\n", now_max, now_max_index);
        break;
      }
    }
    if (complete) return;
```



```c
for (int j = 0; j < fs->FCB_ENTRIES; j++){
  // make sure this is not vacant;
  int vacant = 1;
  for (int n = 0; n < fs->MAX_FILENAME_SIZE; n++){
    if (fs->volume[fs->SUPERBLOCK_SIZE + j*(fs->FCB_SIZE) + n] != 0){
      vacant = 0;
      // printf("没有空啊\n");
      break;
    }
  }
  if (vacant) break;
  int time = (fs->volume[fs->SUPERBLOCK_SIZE + j*(fs->FCB_SIZE) + 24] << 8) + fs->volume[fs->SUPERBLOCK_SIZE + j*(fs->FCB_SIZE) + 25];
  if ((time < prev_max) && (time > now_max)){
    now_max = time;
    now_max_index = j;
  }
}

char name[20];
for (int j = 0; j < fs->MAX_FILENAME_SIZE; j++){
  name[j] = fs->volume[fs->SUPERBLOCK_SIZE + now_max_index*(fs->FCB_SIZE) + j];
  if (name[j] == '\0') break;
}
printf("[fs_gsys] : %s\n", name);
prev_max = now_max;
```

For example, in LS_D, the most fit file is the **most recently modifed file**, which means **modified time is largest**. So look at the modified time in FCB. In the first iteration, set **an upper bound of 0xffff** to ensure **every FCB is qualified** to print out. **Record the largest modified time as the second iteration upperbound**. In the second iteration, only the files with modified time less than the upperbound are qualified to print out. Iteration by iteration, all files can be printed out.

In the bonus part, the procedure is the same, but there is one more restriction: only the files in the current directory can be printed out. Sorry that when implementing this part of bonus, it's near 12pm on November 30th, so I have no time to fix it.

2.3.2 RM

The removal part is simple if the implementation of compaction is successful. The only difference is that there is no need to add the file back after compaction. Other things are the same.

2.3.3 RM_RF

I have no time to implement this part of bonus. I'm so sorry about it. My mind goes like this. Since this bonus requires the depth of directory to be no more than 3, there is no need to use recursive methods to solve it. When the removed directory depth is 1, it's not possible to remove the root directory, so raise an error. When the depth is 2, first check all its children by searching the whole FCB. All children files should be removed. If there are any children directories, the children of them should only be files, so remove them directly using RM. Then remove the children directories, lastly remove itself. When the depth is 3, remove its children files first then itself.

2.3.4 CD and CD_P

CD and CD_P are both simple since the program maintains a global variable to record the current directory. Only updating that variable is fine.

2.3.5 MKDIR

MKDIR is the directory version of `fs_open` . Just need one more step to check whether it is valid. So no more words here.

Part 3: How to Run My Program

1. Make sure you are in the directory: `Assignment_4_120090575\source` or `Assignment_4_120090575\bonus` , type `sbatch ./slurm.sh` in the terminal.

2. You will get a `result.out` file to see the `fs_gsys` output and a `snapshot.bin` file as the dump data. `snapshot.bin` should be the same as `data.bin` in test case 4.

Part 4: Environment

- CUDA version:

  nvcc: NVIDIA (R) Cuda compiler driver

  Copyright (c) 2005-2022 NVIDIA Corporation

  Built on Wed_Jun__8_16:49:14_PDT_2022

  Cuda compilation tools, release 11.7, V11.7.99

  Build cuda_11.7.r11.7/compiler.31442593_0
- GPU: Quadro RTX 4000
- VS Code version: version 1.70
- OS version:
- CentOS Linux release 7.5.1804 (Core)
- kernel version:

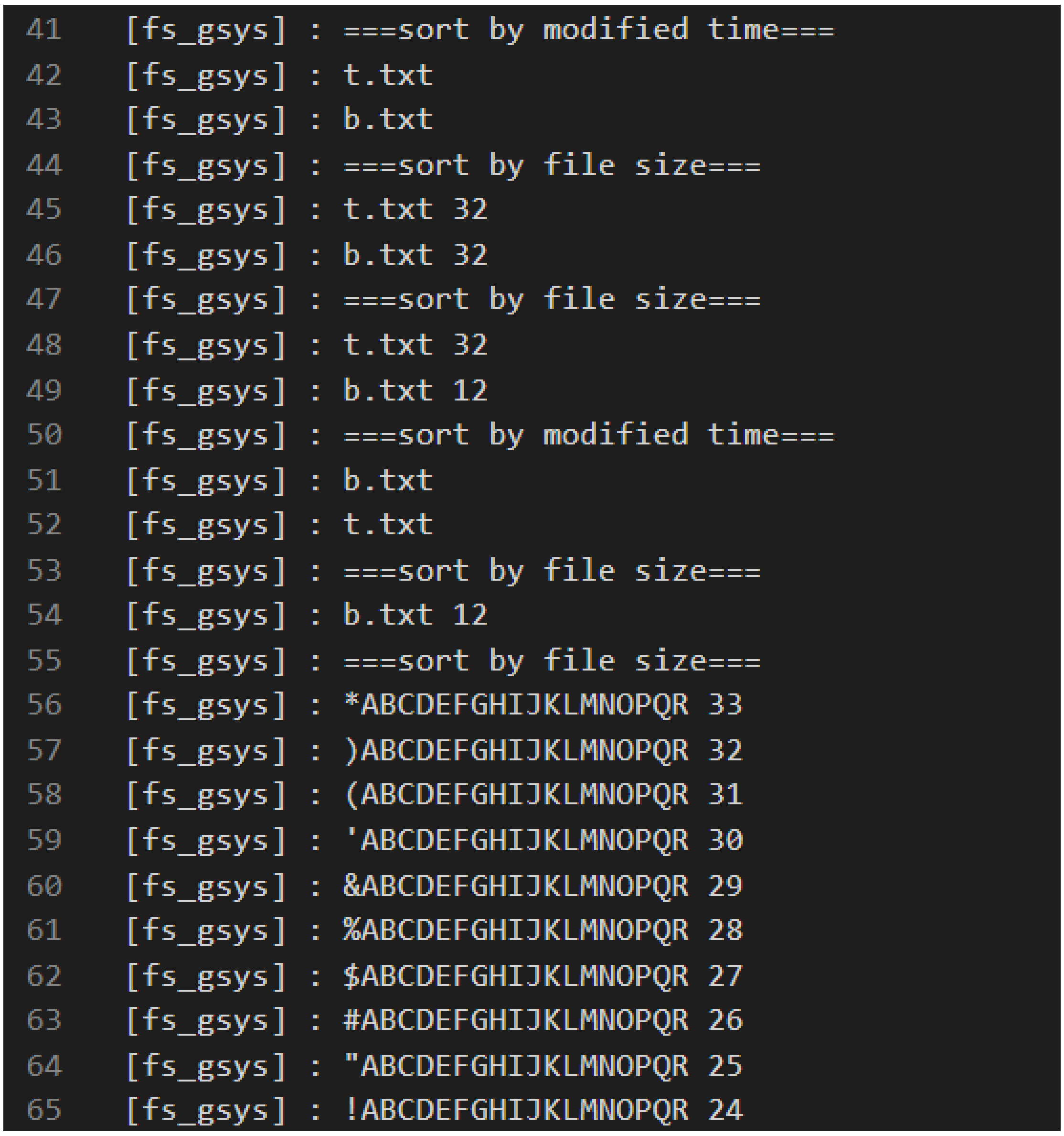
  Linux version 3.10.0-862.el7.x86_64 (builder@kbuilder.dev.centos.org) (gcc version 4.8.520150623 (Red Hat 4.8.5-28) (GCC)) #1 SMP Fri Apr 20 16:44:24 UTC 2018

Part 5: What I've Learnt

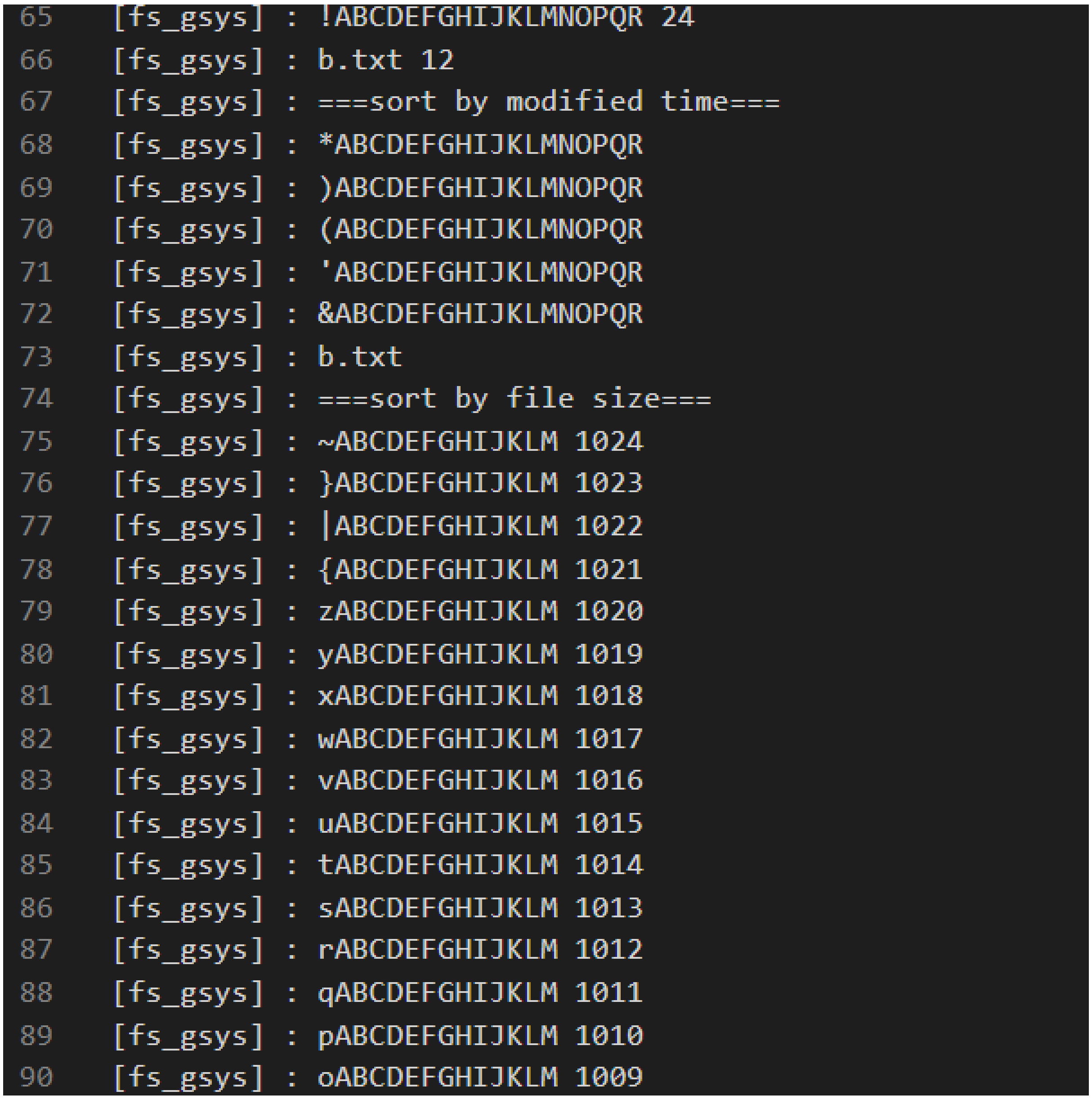
1. The file system management is well learnt and familiar.
2. Some allocation, like contiguous allocation and linked allocation are more familiar, expecially contiguous allocation.
3. The debug skill is enhanced.

Part 6: Screenshot

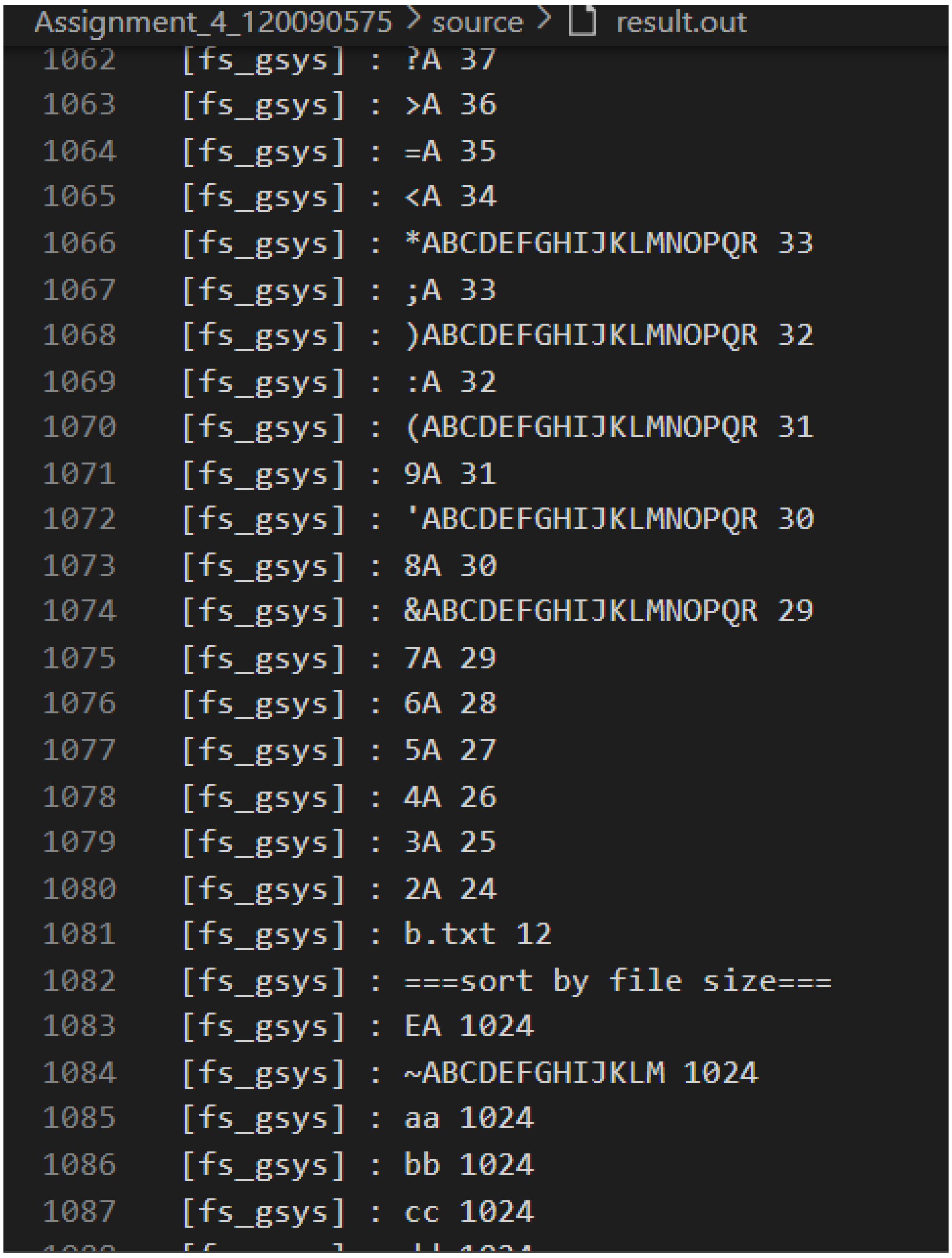
Test case 3:



```
[fs_gsys] : ===sort by modified time===
[fs_gsys] : t.txt
[fs_gsys] : b.txt
[fs_gsys] : ===sort by file size===
[fs_gsys] : t.txt 32
[fs_gsys] : b.txt 32
[fs_gsys] : ===sort by file size===
[fs_gsys] : t.txt 32
[fs_gsys] : b.txt 12
[fs_gsys] : ===sort by modified time===
[fs_gsys] : b.txt
[fs_gsys] : t.txt
[fs_gsys] : ===sort by file size===
[fs_gsys] : b.txt 12
[fs_gsys] : ===sort by file size===
[fs_gsys] : *ABCDEFGHIJKLMNOPQR 33
[fs_gsys] : )ABCDEFGHIJKLMNOPQR 32
[fs_gsys] : (ABCDEFGHIJKLMNOPQR 31
[fs_gsys] : 'ABCDEFGHIJKLMNOPQR 30
[fs_gsys] : &ABCDEFGHIJKLMNOPQR 29
[fs_gsys] : %ABCDEFGHIJKLMNOPQR 28
[fs_gsys] : $ABCDEFGHIJKLMNOPQR 27
[fs_gsys] : #ABCDEFGHIJKLMNOPQR 26
[fs_gsys] : "ABCDEFGHIJKLMNOPQR 25
[fs_gsys] : !ABCDEFGHIJKLMNOPQR 24
```

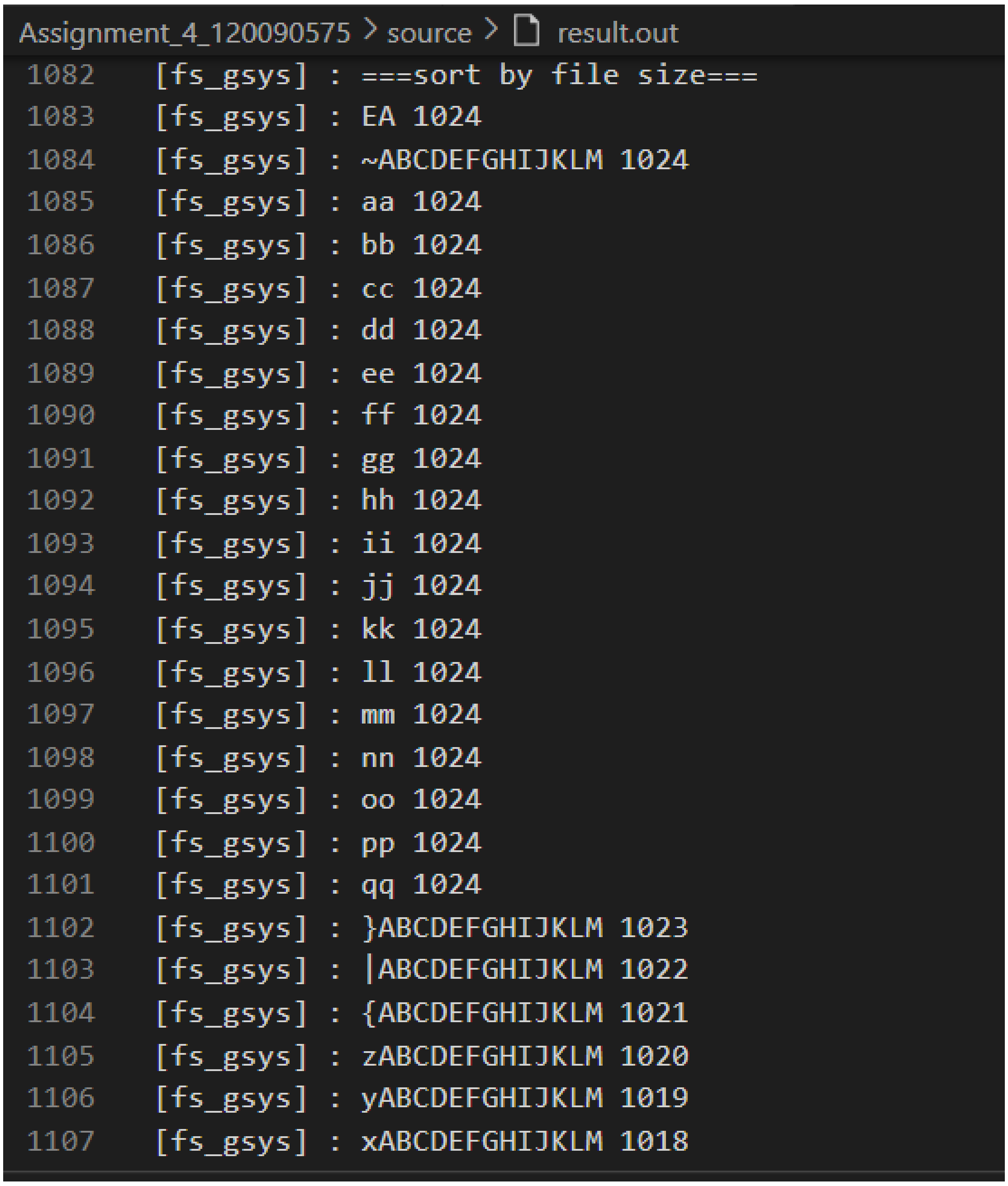


```
[fs_gsys] : !ABCDEFGHIJKLMNOPQR 24
[fs_gsys] : b.txt 12
[fs_gsys] : ===sort by modified time===
[fs_gsys] : *ABCDEFGHIJKLMNOPQR
[fs_gsys] : )ABCDEFGHIJKLMNOPQR
[fs_gsys] : (ABCDEFGHIJKLMNOPQR
[fs_gsys] : 'ABCDEFGHIJKLMNOPQR
[fs_gsys] : &ABCDEFGHIJKLMNOPQR
[fs_gsys] : b.txt
[fs_gsys] : ===sort by file size===
[fs_gsys] : ~ABCDEFGHIJKLM 1024
[fs_gsys] : }ABCDEFGHIJKLM 1023
[fs_gsys] : |ABCDEFGHIJKLM 1022
[fs_gsys] : {ABCDEFGHIJKLM 1021
[fs_gsys] : zABCDEFGHIJKLM 1020
[fs_gsys] : yABCDEFGHIJKLM 1019
[fs_gsys] : xABCDEFGHIJKLM 1018
[fs_gsys] : wABCDEFGHIJKLM 1017
[fs_gsys] : vABCDEFGHIJKLM 1016
[fs_gsys] : uABCDEFGHIJKLM 1015
[fs_gsys] : tABCDEFGHIJKLM 1014
[fs_gsys] : sABCDEFGHIJKLM 1013
[fs_gsys] : rABCDEFGHIJKLM 1012
[fs_gsys] : qABCDEFGHIJKLM 1011
[fs_gsys] : pABCDEFGHIJKLM 1010
[fs_gsys] : oABCDEFGHIJKLM 1009
```



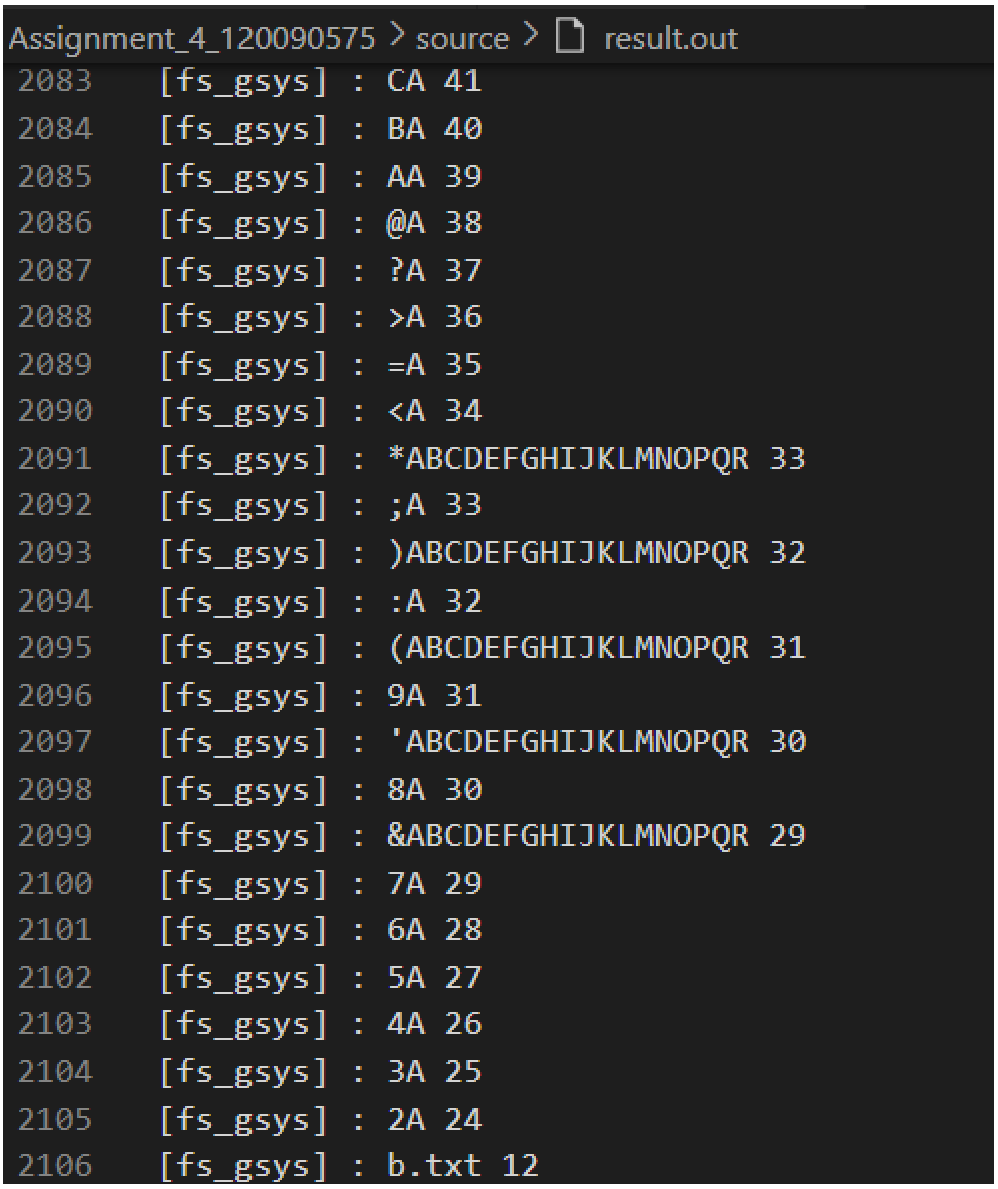
Assignment_4_120090575 > source > result.out

```
[fs_gsys] : ?A 37
[fs_gsys] : >A 36
[fs_gsys] : =A 35
[fs_gsys] : <A 34
[fs_gsys] : *ABCDEFGHIJKLMNOPQR 33
[fs_gsys] : ;A 33
[fs_gsys] : )ABCDEFGHIJKLMNOPQR 32
[fs_gsys] : :A 32
[fs_gsys] : (ABCDEFGHIJKLMNOPQR 31
[fs_gsys] : 9A 31
[fs_gsys] : 'ABCDEFGHIJKLMNOPQR 30
[fs_gsys] : 8A 30
[fs_gsys] : &ABCDEFGHIJKLMNOPQR 29
[fs_gsys] : 7A 29
[fs_gsys] : 6A 28
[fs_gsys] : 5A 27
[fs_gsys] : 4A 26
[fs_gsys] : 3A 25
[fs_gsys] : 2A 24
[fs_gsys] : b.txt 12
[fs_gsys] : ===sort by file size===
[fs_gsys] : EA 1024
[fs_gsys] : ~ABCDEFGHIJKLM 1024
[fs_gsys] : aa 1024
[fs_gsys] : bb 1024
[fs_gsys] : cc 1024
```



Assignment_4_120090575 > source > result.out

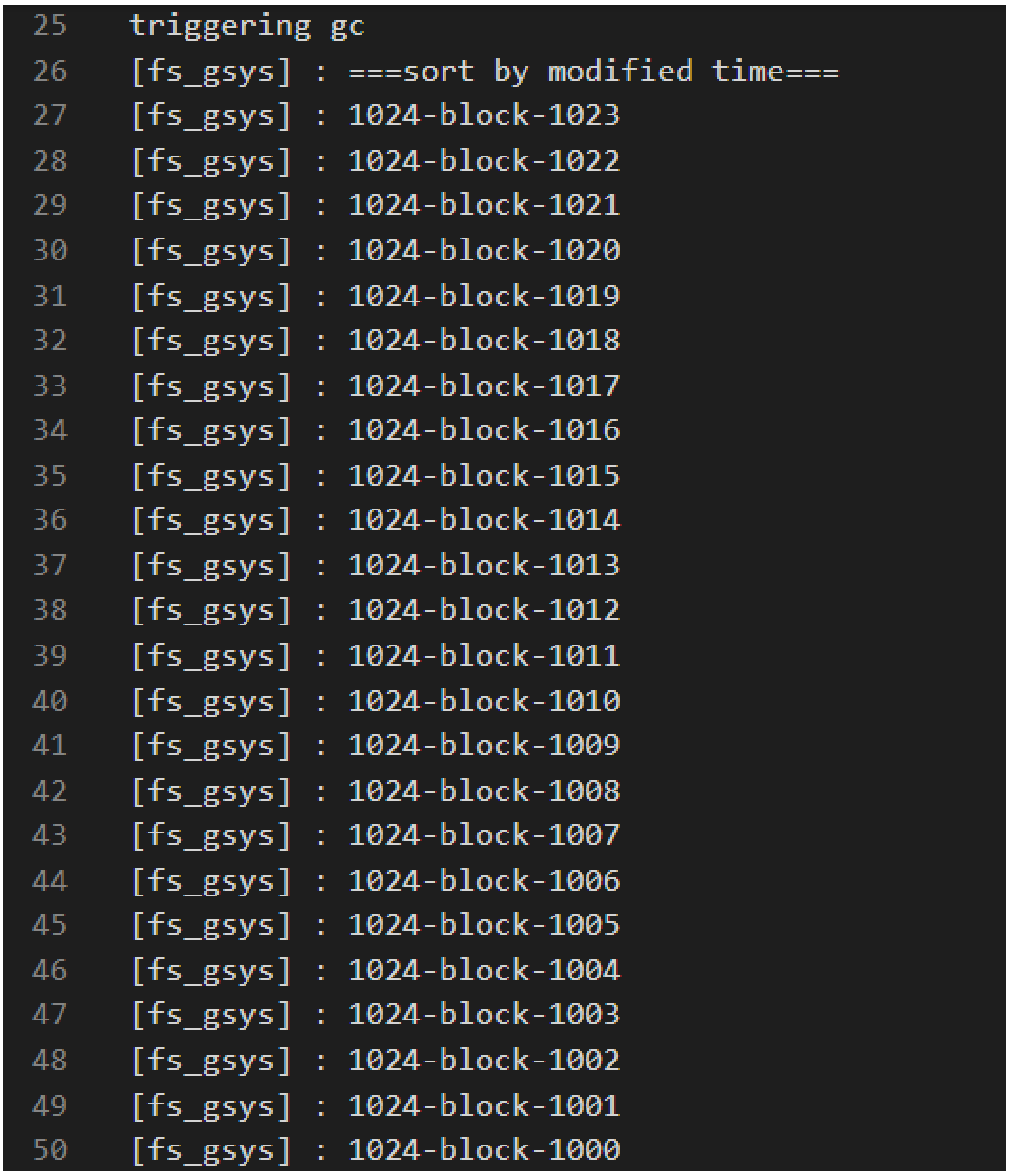
```
[fs_gsys] : ===sort by file size===
[fs_gsys] : EA 1024
[fs_gsys] : ~ABCDEFGHIJKLM 1024
[fs_gsys] : aa 1024
[fs_gsys] : bb 1024
[fs_gsys] : cc 1024
[fs_gsys] : dd 1024
[fs_gsys] : ee 1024
[fs_gsys] : ff 1024
[fs_gsys] : gg 1024
[fs_gsys] : hh 1024
[fs_gsys] : ii 1024
[fs_gsys] : jj 1024
[fs_gsys] : kk 1024
[fs_gsys] : ll 1024
[fs_gsys] : mm 1024
[fs_gsys] : nn 1024
[fs_gsys] : oo 1024
[fs_gsys] : pp 1024
[fs_gsys] : qq 1024
[fs_gsys] : }ABCDEFGHIJKLM 1023
[fs_gsys] : |ABCDEFGHIJKLM 1022
[fs_gsys] : {ABCDEFGHIJKLM 1021
[fs_gsys] : zABCDEFGHIJKLM 1020
[fs_gsys] : yABCDEFGHIJKLM 1019
[fs_gsys] : xABCDEFGHIJKLM 1018
```



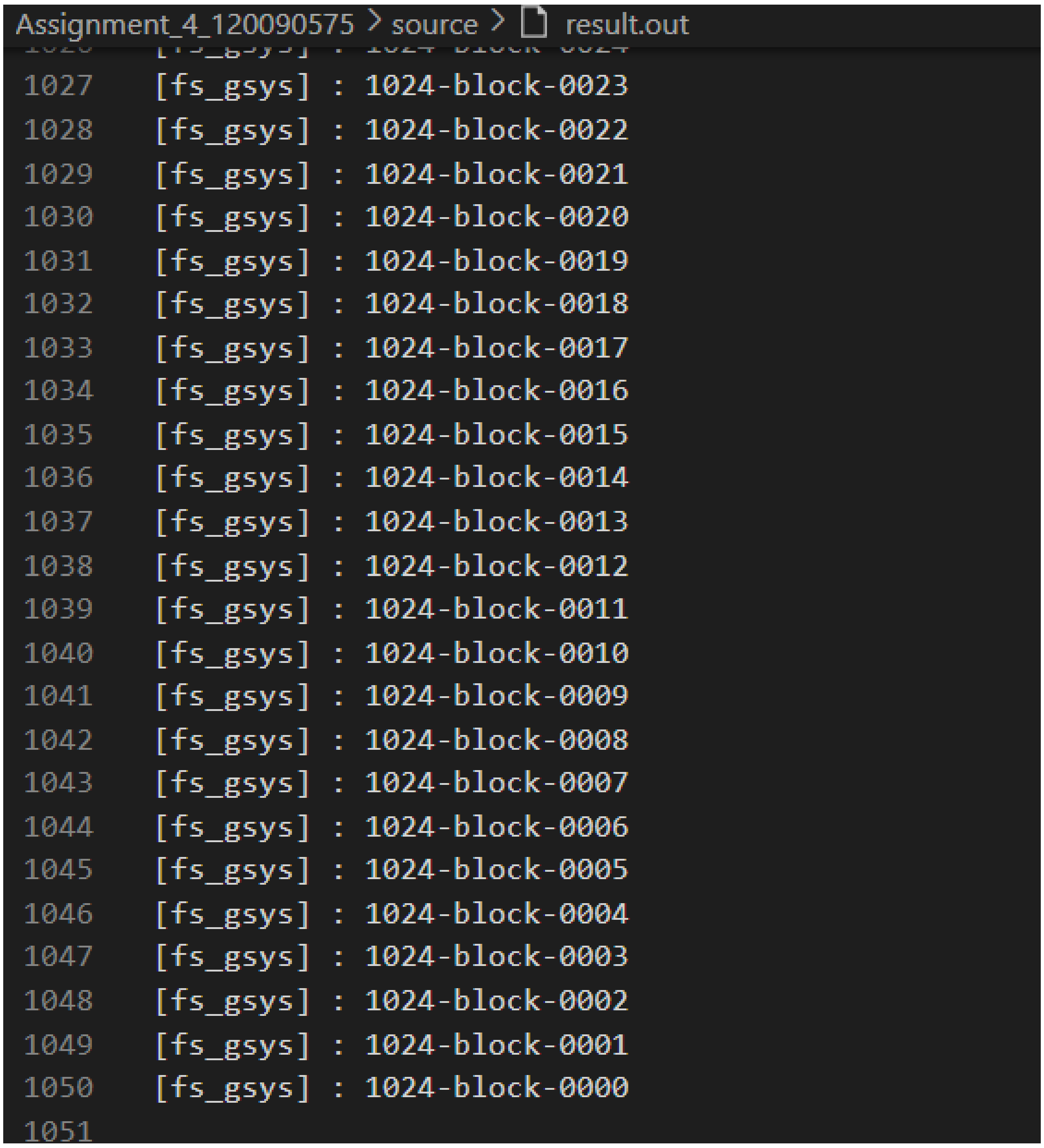
Assignment_4_120090575 > source > result.out

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

Test case 4:



```
triggering gc
[fs_gsys] : ===sort by modified time===
[fs_gsys] : 1024-block-1023
[fs_gsys] : 1024-block-1022
[fs_gsys] : 1024-block-1021
[fs_gsys] : 1024-block-1020
[fs_gsys] : 1024-block-1019
[fs_gsys] : 1024-block-1018
[fs_gsys] : 1024-block-1017
[fs_gsys] : 1024-block-1016
[fs_gsys] : 1024-block-1015
[fs_gsys] : 1024-block-1014
[fs_gsys] : 1024-block-1013
[fs_gsys] : 1024-block-1012
[fs_gsys] : 1024-block-1011
[fs_gsys] : 1024-block-1010
[fs_gsys] : 1024-block-1009
[fs_gsys] : 1024-block-1008
[fs_gsys] : 1024-block-1007
[fs_gsys] : 1024-block-1006
[fs_gsys] : 1024-block-1005
[fs_gsys] : 1024-block-1004
[fs_gsys] : 1024-block-1003
[fs_gsys] : 1024-block-1002
[fs_gsys] : 1024-block-1001
[fs_gsys] : 1024-block-1000
```



```
[fs_gsys] : 1024-block-0023
[fs_gsys] : 1024-block-0022
[fs_gsys] : 1024-block-0021
[fs_gsys] : 1024-block-0020
[fs_gsys] : 1024-block-0019
[fs_gsys] : 1024-block-0018
[fs_gsys] : 1024-block-0017
[fs_gsys] : 1024-block-0016
[fs_gsys] : 1024-block-0015
[fs_gsys] : 1024-block-0014
[fs_gsys] : 1024-block-0013
[fs_gsys] : 1024-block-0012
[fs_gsys] : 1024-block-0011
[fs_gsys] : 1024-block-0010
[fs_gsys] : 1024-block-0009
[fs_gsys] : 1024-block-0008
[fs_gsys] : 1024-block-0007
[fs_gsys] : 1024-block-0006
[fs_gsys] : 1024-block-0005
[fs_gsys] : 1024-block-0004
[fs_gsys] : 1024-block-0003
[fs_gsys] : 1024-block-0002
[fs_gsys] : 1024-block-0001
[fs_gsys] : 1024-block-0000
```