CSC3150 Assignment4

119010054 Dai Yulong

1. Run Methods

My project adopts the recommended environment on TC301 computer:

For both two programs (`Source` and `Bonus`), I write makefile to compile it. So only two steps are needed to run the program.

1. Using `cd` to enter the `source` or `bonus` directory, then type `make` command and enter.
2. Enter `./main.out` to execute the program:

```
[cuhksz@TC-301-18 Source]$ make
nvcc --relocatable-device-code=true main.cu file_system.cu user_program.cu -o
main.out
```

```
[cuhksz@TC-301-18 Bonus]$ make
nvcc --relocatable-device-code=true main.cu file_system.cu user_program.cu -o
main.out
```

3. In the corresponding directory, type `make clean` command to clear the `main.out`.

2. Design Ideas

2.1 Overall ideas

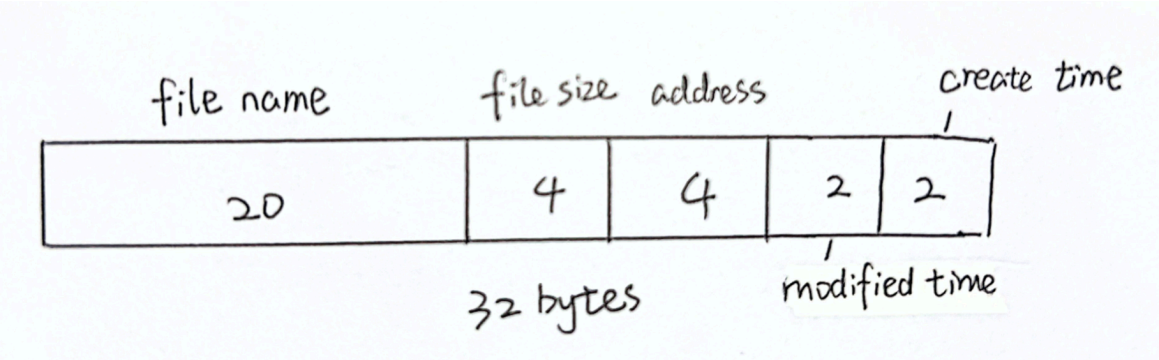
In my program, I mainly develop three components of the file system. The first one is the file control blocks (FCB), which is used to stored basic attribute of one file (in bonus part, the basic information of directory also stored here) and the pointer points to the physical content address (the number of blocks) of one file. The second component is the physical memory where stores the contents of files, whose basic storing unit is one block (32B). The third component is the super block which stores the 4KB bit-vector where each bit in it indicates whether one corresponding block in the physical memory is occupied. Belows are my design details.

2.2 Design ideas of `Source`

2.2.1 Design of FCB blocks

In the file control blocks region, there are totally 1024 FCBs where each size is 32B. One FCB stores one file's basic information, including file name (20 bytes), file size (4 bytes), starting address (number of head of its blocks in physical memory space: 4 bytes), create time (2 bytes), modified time (2 bytes). The following figure shows the designed structure of one FCB.

Figure 1. The Design of `Source` FCB

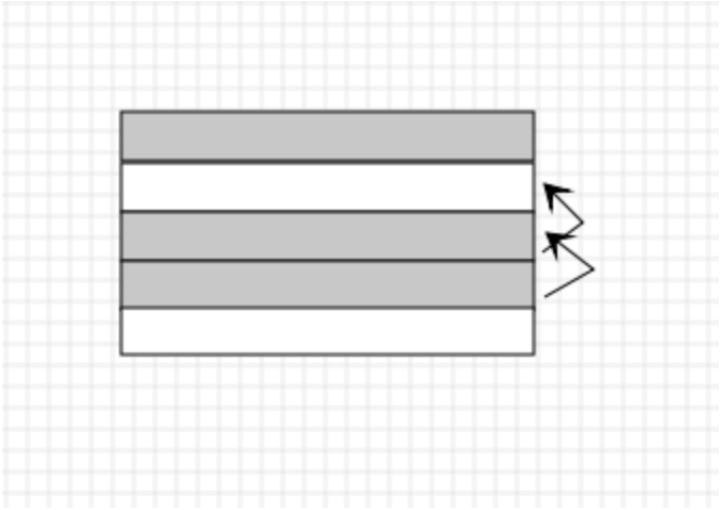


Notice that file size and address are stored in `int` type, create time and modified time are stored in `short` type.

2.2.2 Design of Physical memory

As for the physical memory area where the file contents are stored, it is divided into 32*1024 blocks, each block occupying 32B. The most important issue for this area is maintenance, including ensuring contiguous allocation, compaction, etc.In terms of contiguity, the blocks allocated to a file in this area should be contiguous to avoid external segments. More importantly, to make good use of memory, there is a mechanism to do compaction as soon as a free block becomes available. Figure 2 shows how this system solves the compression problem. If a file frees a free block, the system will first identify the size of this area by reading the size attribute stored in the FCB. Then, it will update the bitmap stored in the super block area. Finally, based on the bit map, it will be easier to move the blocks behind forward to occupy the empty space.

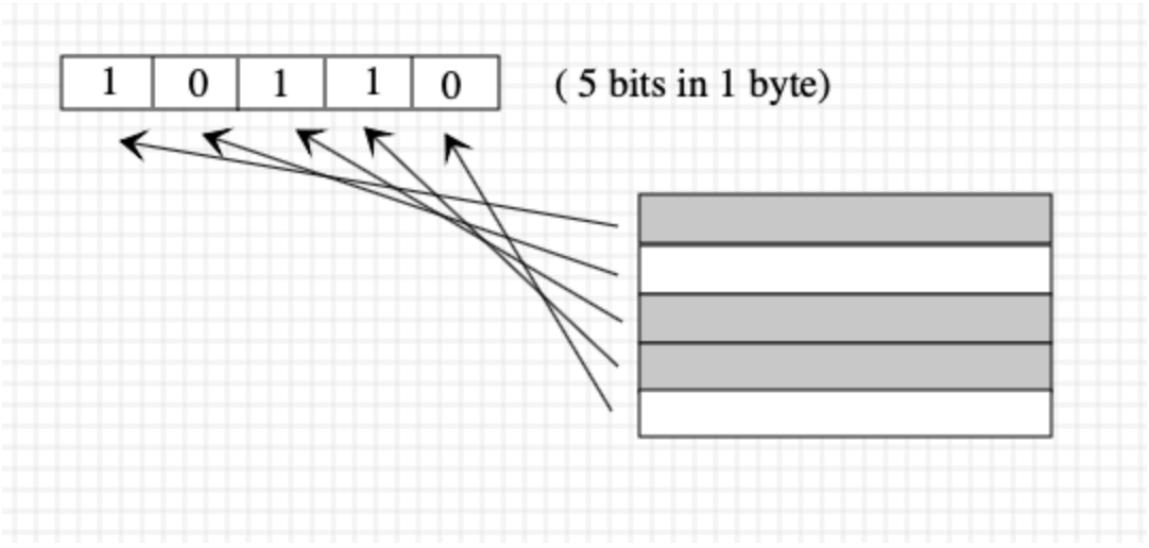
Figure 2. The Compaction Machnism (Move blocks behind to occupy the empty block)



2.2.3 Design of Super block (bit map area)

In super block area, a bitmap is designed to indicate whether a block in physical memory is occupied or not. There are a total of 4KB bits to indicate whether the corresponding block is occupied or not. When the program starts, all bits will be initialized to 0. When the block is allocated to a file, the corresponding bit will be modified to 1 (and then to 0 if it is freed by a file). Here I use bit operation to update the bitmap value. For example, `fs->volume[(start_block + i) / 8] = fs->volume[start_block / 8] | (1 << ((start_block + i) % 8))` is used to write the corresponding bit to 1. Figure 3 shows how the bits in the bitmap are mapped to the blocks in the content area.

Figure 3. The Bitmap Implementation



2.2.4 Implementaion of APIs

In the `Source part`, totally five functions are implemented, including `fs_open` , `fs_write`, `fs_read`, `fs_gsys(RM)` , `fs_gsys(LS_S\LS_D)`.

1. **`fs_open(Filesystem \*fs, char \*s, int op)`**

This function will open a file and return a pointer to the FCB of that file. If the file does not exist, it will create a file, allocate an empty block to it, and don't set its content. First, it will search for all valid FCBs and check if their names are the same as the file name. If it finds an FCB with the same name as the file, it will return the address of this FCB. And if it does not find the file, it will create a new file by the following process (under write mode).

- Firstly, find empty blocks for new files. Because we have already compacted the content, then the number of blocks found will certainly meet the requirement.
- Secondly, find a new FCB. Beacuse we also compact the FCB blocks, the empty block will srictly follow the existence files.
- Then, initialize all informations in FCB blocks, including, creation time, modification time (when first created, its modification time will be the creation time), size (initially 0) and the address of the block we found in the first step.
- Finally, return the address of the newly created FCB.

2. **`fs_read(Filesystem \*fs, uchar \*output, u32 size, u32 fp):`**

   In this API, we use fp directly to locate the FCB and retrieve the address of the content area from the FCB, and then read the data into the buffer one by one.

3. **`fs_write(Filesystem \*fs, uchar \*input, u32 size, u32 fp)`**

   This function uses a pointer to locate the FCB. There're totally three situations when writing data:

   - When the located FCB is empty, directly write data into the physical memory (compaction ensures there is always enough space for writing data < 1024 bytes). Then update the file size in FCB block. And also update the bitmap.
   - When the located FCB is not empty but occupies the same number of blocks as new written data, no need for compaction. Directly delete old data and write new data in. Update the file size in FCB block. No need for updating the bitmap.
   - When the located FCB is not empty and the number of used blocks not equals (increase or decrease), we need to reallocate the content. In my program, I first delete the original file content using `fs_gsys(RM)`, then using `fs_open()` to find a new place for FCB. After the reallocation, the file will have enough spaces for the number of blocks needed in the input buffer. Finally, we write the bytes in the input buffer to the blocks in the content area one by one. Also remember to update the bitmap and file size, modified time, created time and starting address in the FCB.

4. **`fs_gsys(RM):`**

In this function, first, it will find the corresponding FCB based on the name of the file. After the FCB is found, it will first retrieve the address of the block header in the content area and the size of the file. After the FCB is created, it will first retrieve the address of the block header in the content area and the size of the file. We store this information before deleting the file because it is needed when performing compression.Then, it clears everything in the corresponding FCB, bitmap and content. Finally, it will use the size and address of the block for compaction. The brief mechanism is shown in Figure 2. It will first compact FCB blocks (just for easier sort). Notice that it needs to update the FCB because some blocks are moved forward during the compaction process. Then it will compact the bitmap. Finally, compact the file content base on the bitmap.

5. `fs_gsys(LS_D / LS_S):`

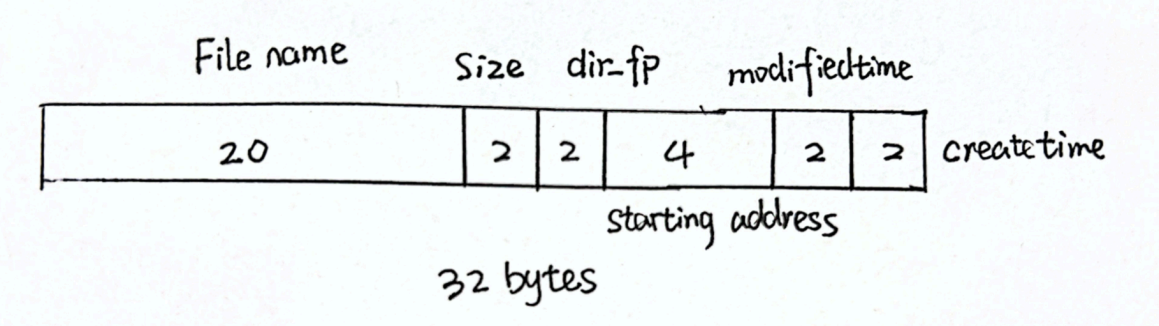
   I use insertion sort to sort all files. This algorithm shows great performance with low additional space cost (only need one array). Notice that when sorting the files by size (size equals, sorting by creating time), two inserting sort procedure are needed.

2.3 Design ideas of `Bonus`

2.3.1 Design of FCB blocks

There's a slightly difference between design of `Source` FCB and `Bnous` FCB. Notice that each file or directory are considerd the same to be stored in one FCB. One FCB stores one file's basic information, including file name (20 bytes), file size (2 bytes), directory pointer (2 bytes), starting address (number of head of its blocks in physical memory space: 4 bytes), create time (2 bytes), modified time (2 bytes).The following figure shows the designed structure of one FCB.

Figure 4. The Design of `Bonus` FCB



Notice that starting address is stored in `int` type. File size, directory pointer, create time and modified time are stored in `short` type. Actually the maximum number of file size is 1024. It's suitable use a `short` integer to store the value. The directory pointer points to its parent directory (**1025** when it's the root).

There is a global variable `dir` stores the index of parent directory (also has a FCB). And for a directory FCB,

- The size is the sum of character bytes of all files name (include subdirectories).
- It doesn't have corresponding bitmap or content.
- When its files or subdirectories are updated (write or delete), the modified time of this directory will be updated.
- The starting address of directory is initialized as 32769, which is used to distinguish directory from files.

The design of bitmap and physical memory design are the same as `Source`.

2.3.2 Implementaion of APIs

For those `fs_open` , `fs_write`, `fs_read`, `fs_gsys(RM)` , `fs_gsys(LS_S\LS_D)` functions, there's only slight different. Notice that in `Bonus` function, I don't compact FCB blocks, just compact file content and bitmaps. The `Bonus` newly implements some new APIs, including `fs_gsys(CD)` , `fs_gsys(CD_P)`, `fs_gsys(RM_RF)`, `fs_gsys(PWD)`, `fs_gsys(LS_D)`.

1. **`fs_gsys(MKDIR)`**

   To create a new directory, I modified the `source` code section implementation of `fs_open( )` to support directory creation as well. Like creating a file, creating a directory starts by finding an empty FCB and storing the information in the FCB.

2. **`fs_gsys(CD)`**

   Since I use global variable `dir` to store the current directory. The CD implementation first searches for target FCB in all FCBs and then assigns the new address (index) of the FCB to `dir`.

3. **`fs_gsys(CD_P)`**

   Because the global variable `dir` points to current directory. The program finds the current directory's parent directory, then assigns it to `dir`.

4. **`fs_gsys(RM_RF)`**

In this part, the program first searches the target directory (ensure it's a directory and not a file). Then use two for loops to literately delete files and directories. The delete order is first the bottom files, then its directory, then files that in the same layers (if needed), then its directory(if needed). Notice that we also need to update the size of corresponding parent directory.

5. **`fs_gsys(PWD)`**

   With `dir` (points to current directory), we can find the directory FCB for the current directory. Since the directory FCB also stores a pointer to its parent directory, we can use it to further find its parent directory until its parent is the root directory.

6. **`fs_gsys(LS_D / LS_S)`**:

   The principle is similar to the one implemented in the `source` code section. The difference is that these will find the current directory block and narrow down the comparison between the file addresses in it.

3. Screenshot of sample output

3.1 The screenshot of test case 1 (basic version)

```
[cuhksz@TC-301-18 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
```

3.2 The screenshot of test case 2 (basic version)

```
[cuhksz@TC-301-18 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
```

3.3 The screenshot of test case 3 (basic version)

This is the tail of part 3 output:

```
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
[cuhksz@TC-301-18 Source]$
```

3.4 The screenshot of bonus

```
[cuhksz@TC-301-18 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 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
```

4. The problems I met and my solution

Belows are my problems and corresponding solutions when implementing file system:

- The first problem I met is how to store the FCB informations in aninformation unsigned character array. Because FCB information, such as size, starting address, are integer. And if we use bit operations to store the integers in an unsigned character array, e.g. using four unsigned characters to store one `int` integer, the program codes will be too complex and bloated, which greatly reduce the code readability. So I want to find another way to achieve this

transformation. And I think of cast between different pointer types. For example:

```
*(short *)&fs->volume[fs->SUPERBLOCK_SIZE + num * fs->FCB_SIZE + fs->MAX_FILENAME_SIZE + 8] = gtime;
```

Here I use short type integer to store the modified time of file. By direct casting `char` type to `short` type, the complier will regard the two positions in it as short integer. Notice that `short` type is 2 bytes. `uchar` type is 1 byte. Thus, a `short` type pointer occupies two positions in the `uchar` array.

- The second problem is about sort algorithm. In this project, we need to sort the files by modified time or file size. Firstly, I directly use three for loops to sort the files. I find that it takes too long to sort all 1024 files (in case 3). Then I consider use three additional arrays, which greatly reduce the time complexity but increase lots of space. Finally, I consider use insertion sort to sort all files. This algorithm shows great performance with low additional space cost (only need one array). Notice that when sorting the files by size (size equals, sorting by creating time), two inserting sort procedure are used.

5. Learning

After finishing the assignment4, I have an overall and deep understanding of file system.

- I have a general idea about how to implement open, read, and write operations in a file system . The most significant thing is to maintain the attributes in FCB part. Then we can get much information about these files. When a file is open, the first thing is to find the file in the FCB. If we want to write or read this file, we also use fp to find the address where data are stored. And fp is the address of that file in FCB.
- I understand the bit-map method when indicate the status of some blocks. This method maps a block with a bit. Then we can use just one bit to judge whether the block is occupied. This method saves much memory space to control the status of something. It is a powerful tool when we want to control a large volume of memory.
- I know several methods to reduce the external fragment. I can use contiguous allocation to allocate a large area without internal fragment. I can also use compaction to compact the storage area to free space for larger files, which is implemented in the program.