CSC3150 Assignment 4 Report

119010177 Muhan Lin

I. Introduction

In this project, a file system of operating system is implemented on GPU. The global memory of GPU is taken as a volume from a hard disk. This volume consists of super block (bit map), File Control Block (FCB) and storage. The input and output of this file system are data.bin and snapshot.bin respectively. After starting the program, CUDA Kernel is launched and CPU writes data to input array. When the program exits, CPU loads data from output array into snapshot.bin.

II. Design and Problem Solution

1. File System Strategy

The file system consists of three parts:

- super block (bit map)
- FCB
- storage

Each bit of the bit map records the occupation situation of each block. 1 means "occupied", while 0 means "empty". FCB records the information of each file. In this project, FCB records the name, size, starting address, modification time and parent (in bonus) of files. Storage part stores the content of files.

When a new file is opened with `fs_open()`, the file system needs to find large enough space to store it with bit map, because the space allocated must consists of integer number of blocks. When the space is found, the corresponding bits of bit map will be set. Then the information of this file will be recorded by FCB and the content of this file will be written to the space. The file system returns the base address of the FCB entry recording this file as the file pointer so that the program can get access to not only the file content but also all information about this file.

Correspondingly, when the file system removes a file, the concerned super block bits, FCB entry and storage blocks need to be cleared. The bytes in the corresponding FCB entry will be written as 0xFF, meaning DELETED. This thinking is similar with what is used by open addressing in hashing table. In this way, when searching for an empty space in FCB, any entry whose every byte is 0x00 (initial setting) means the file searched for is not in this file, because FCB entries are used in order. But when inserting a file into FCB, entry marked DELETED can be used again. Both space and time consumption of seaching FCB are reduced.

What is worth noticing here is compaction strategy. To avoid external fragmentation, the file system stores all files in consecutive blocks all the time. (The blocks are used in succesive order when allocating space to new files.) But in some cases, empty space appears between different files. Then the file system needs to move all the files behind this empty space forwards to fill it. This is called compaction. When compacting, the file system will clear and re-mark bits in the bit map. The size and starting address records in FCB will also be revised, but modification time record is remained.
In this project, there are two cases where compaction is applied:

- Once after a file is deleted.
- Increasing the block number of a file

The reason of the first case is straight-forward. The reason behind the second case is as follows. Before size increasing, all files are stored consecutively. If one file is enlarged, its original position cannot contain it any more. Because all data of a file must be stored in successive addresses, the file system needs to find another larger space for this file. Therefore, the space originally occupied by this file becomes spare. The files behind this space need to be moved forward to fill this space.

In the bonus part, the file system can group files in a tree-structured directory. Each directory is a file, whose data, stored in storage as well, are names of all files and subdirectories in it. Each file and directory can obtain its parent fp from FCB. There is a global variable `crtPathFp` recording the address of the current directory FCB entry.

2. Volume Structure Design and Implementation

In this project, the volume is implemented as an uchar array, so the space of any part is organized in bytes. The advantage is that the array index is the same as the address.

(a) Storage

This part is used to store file contents, whose size is 1024KB, because the maximum file size is 1024 bytes and there are at most 1024 files. As the block size is 32 bytes, there are $1024 * 1024 / 32 = 2^{15}$ blocks.

(b) Super Block

This part is of $4KB = 2^{17}$ bits. Each bit corresponds to one block. This program provides some operation interface functions of super block:

```
__device__ void RecordOccupied(FileSystem *fs, int startIndex, int num);
__device__ void ClearOccupied(FileSystem *fs, int startIndex, int num);
```

These functions are used to set or reset bits in super block starting from `startIndex`th bit (block in storage) and lasting for `num` bits (blocks in storage). The program defines a mask consisting of `num` 1s, shifts the mask to position determined by `startIndex` and set or reset the bits with `|` or `&= ~`, which are common operation for bit manipulation. When doing these operations, the bits of shifting as well as data type size are significant for correct results.

```
__device__ u32 LocateSpace(FileSystem *fs, int size);
```

This function is used to locate a space of given size. The program defines a mask consisting of `size` 1s. `&` operation is used to seach for consecutive 0s when shifting the mask along the super block entries. The problem is that As `size` can be any number from 1 to 32. It seems that checking 4 super block entries (32 bits) at one time is enough for 32 blocks of empty space. However, because each time the mask can only be shifted forwards by integer number of blocks (otherwise the operations will be tedious), if `size` is larger than 24 bits and the spare space is unaligned, checking 4 entries at one time can ignore some vacant bits.

Therefore, `uint64_t` data type is needed to store five entries (five bytes) of bit map each time and get involve in the bit operation.

(c) FCB

FCB has 1024 entries. Each entry is for one file and has 32 bytes. To save space, the usage of these 32 bytes is in unit of bits as follows.

byte[bit]	content
0~19	name
20[7:0],21[7:6]	size
21[5:0],22[7:0],23[7]	starting address
24~28	modification time
29[7:0],30[7:6]	parent

Because the maximum file size is $1024=2^{10}$ bytes, 10 bits are enough for recording size. There are 2^{15} blocks in the storage, and each file must take integer number of blocks, so 15 bits are enough to represent the starting address. The transformation between recorded starting address and the real starting address should be `recorded = (real - SUPER_BLOCK_SIZE - FCB_SIZE) >> 5`. The modification time is recorded by global variable `gtime`, which increases by 1 whenever an open, read or write operation is executed. `parent` is the fp of parent directory. As there are 2^{10} FCB entries, 10 bits are enough to record fp. The transformation should be `recorded = (real - SUPER_BLOCK_SIZE) >> 5`.
Several functions were implemented for recording and checking file information. The bit operations involved are similar to what have been stated in super block part.

```
__device__ u32 checkSize(FileSystem *fs, u32 fp);
__device__ void modifySize(FileSystem *fs, u32 fp, u32 newSize);
__device__ u32 checkStartingAddr(FileSystem *fs, u32 fp);
__device__ void modifyStartingAddr(FileSystem *fs, u32 fp, u32 newAddr);
__device__ uint64_t checkModifyT(FileSystem *fs, u32 fp);
__device__ void modify_ModifyT(FileSystem *fs, u32 fp, uint64_t newT);
__device__ uchar* checkName(FileSystem *fs, u32 fp);
```

A function of searching FCB was also implemented. If the file is found to not in FCB, it will be inserted by this function. The return is fp, the real base address of the FCB entry, convenient for later usage. Name is the comparison standard when searching. Improved linear search is applied, whose strategy has been mentioned before. To reduce time consumption, the program will insert the file into the first empty space encountered and stop the search. DELETED entries will not be used to insert until the search reaches the end of FCB, ie. the file is not in FCB and there is no empty entry in FCB, to ensure each possible entry of the given file has been checked. A variable `delPos` stores the address of the first DELETED entry, so the program can find the first DELETED entry and add the file to FCB in O(1) after it reaches the end of FCB. The newly added file must was just opened and has not been written, so this function records its size as 0. Modification time is also updated. Starting address is recorded as 0xFFFF, meaning not in storage (0 size).

```
__device__ u32 search_FCB_entry(FileSystem *fs, char* s, int op);
```

3. File Operation

a. Open

Actually `search_FCB_entry` completes most work of `fs_open`. Only an extra consideration of `op` is needed. If the mode is to write, the program updates the modification time and clears the file's content in storage for later writing. Because clearing size to 0 requires re-allocation in later work, which is time-expensive, the program keeps the original size and starting address unchanged, and modifies them in `fs_write` if necessary. In this way, if the new size is not larger than the original size, the program will save the time of searching for and moving the file to larger empty space, which is unavoidable if size is modified to 0 once a file is opened to be written.

b. Read

Use pre-defined function `checkStartingAddr(fs, fp)` to get the starting address with fp directly, and copy the data there to output buffer byte by byte, as the storage and buffer are both uchar array.

c. Write

First, use pre-defined function `checkSize(fs, fp)` to get the original file size with fp directly. Compute and compare the original and new block number of this file. If the newly required blocks are not more than the origianl allocated ones, the file remains at the original place. Otherwise, the program first removes the file from storage and compacts, and then searches for larger space with pre-defined `LocateSpace(fs, size)` in bit map to get the new fp. First compact is to prepare more available space. Then the program updates bit map as well as the size, starting address and modification time in FCB correspondingly.

d. LS_S/LS_D

The insertion sort is applied in this routine. The problem is how to recored the files and their properties in an array for efficient checking and searching. The solution is to use an `u32` array `records` to store `fp` of each file. When comparing two files' size or modification time or printing, the program calls pre-defined function to check the desired information of these files with `fp` read out from the array. The swap of two array item is also simple: just to swap two `u32` number. The method of recording and storing file pointer is efficient both for time and space consumption.
In bonus part, the recorded files should not be all files, but should be the files obtained from the data of the current directory.

e. RM

The strategy has been described in "File System Strategy" part.

f. MKDIR

Create a new file with the given directory name with operation similar with `fs_open`. Records its parent in FCB. Add this file's name to its parents' data.

g.CD and CD_P

Update the global variable `crtPathFp` mentioned before. For CD, check whether the given one is a subdirectory of the current directory by searching it in the current directory's storage first.

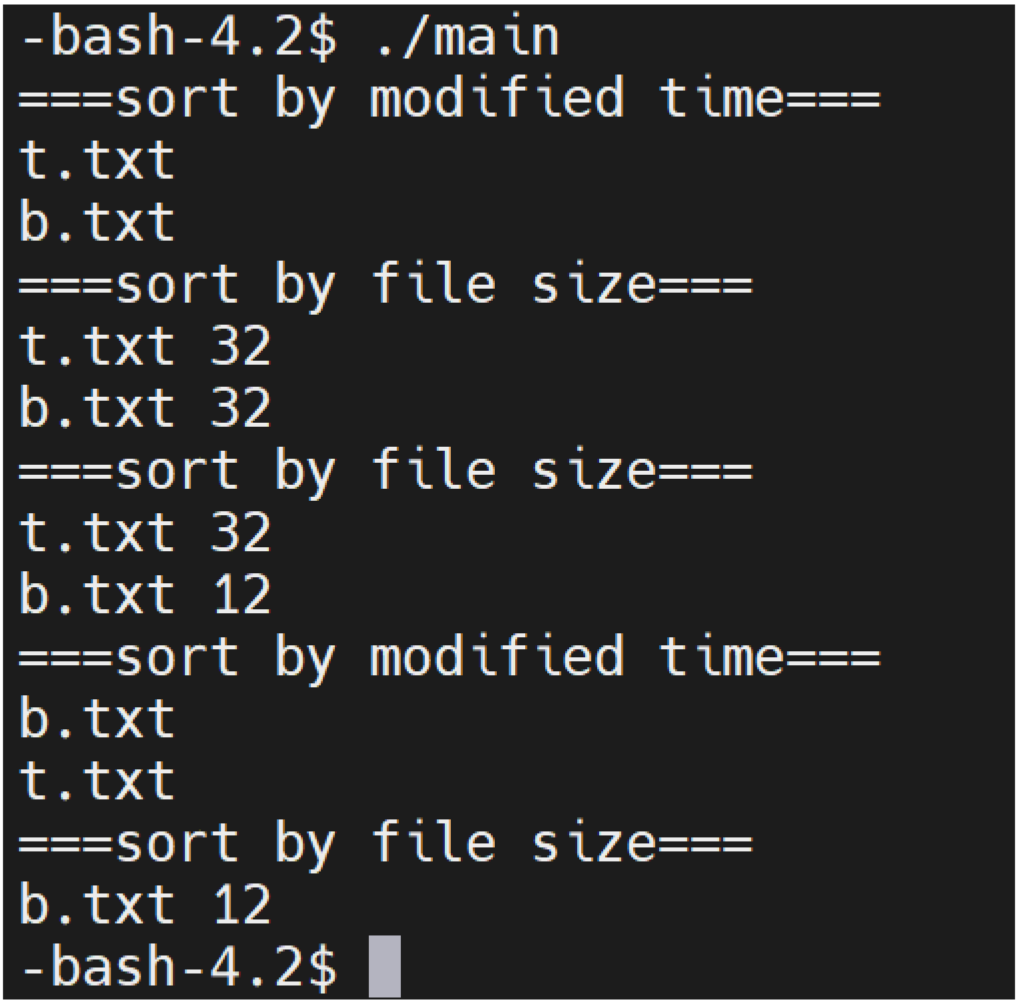
III. Execution Step

```
-bash-4.2$ cd Assignment4_119010177/Source
-bash-4.2$ make
-bash-4.2$ ./main
```

```
-bash-4.2$ cd Assignment4_119010177/Bonus
-bash-4.2$ make
-bash-4.2$ ./main
```

IV. Screenshot

1. test 1



1. test 2

```
===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
```

1. test 3

```
-bash-4.2$ ./main
===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
===sort by file size===
~ABCDEFGHIJKLM 1024
}ABCDEFGHIJKLM 1023
|ABCDEFGHIJKLM 1022
{ABCDEFGHIJKLM 1021
zABCDEFGHIJKLM 1020
yABCDEFGHIJKLM 1019
xABCDEFGHIJKLM 1018
wABCDEFGHIJKLM 1017
vABCDEFGHIJKLM 1016
uABCDEFGHIJKLM 1015
tABCDEFGHIJKLM 1014
sABCDEFGHIJKLM 1013
rABCDEFGHIJKLM 1012
qABCDEFGHIJKLM 1011
pABCDEFGHIJKLM 1010
```

```
pABCDEFGHIJKLM 1010
oABCDEFGHIJKLM 1009
nABCDEFGHIJKLM 1008
mABCDEFGHIJKLM 1007
lABCDEFGHIJKLM 1006
kABCDEFGHIJKLM 1005
jABCDEFGHIJKLM 1004
iABCDEFGHIJKLM 1003
hABCDEFGHIJKLM 1002
gABCDEFGHIJKLM 1001
fABCDEFGHIJKLM 1000
eABCDEFGHIJKLM 999
dABCDEFGHIJKLM 998
cABCDEFGHIJKLM 997
```

```
bABCDEFGHIJKLM 996
aABCDEFGHIJKLM 995
`ABCDEFGHIJKLM 994
_ABCDEFGHIJKLM 993
^ABCDEFGHIJKLM 992
]ABCDEFGHIJKLM 991
\ABCDEFGHIJKLM 990
[ABCDEFGHIJKLM 989
ZABCDEFGHIJKLM 988
YABCDEFGHIJKLM 987
XABCDEFGHIJKLM 986
WABCDEFGHIJKLM 985
VABCDEFGHIJKLM 984
UABCDEFGHIJKLM 983
TABCDEFGHIJKLM 982
SABCDEFGHIJKLM 981
RABCDEFGHIJKLM 980
QABCDEFGHIJKLM 979
PABCDEFGHIJKLM 978
OABCDEFGHIJKLM 977
NABCDEFGHIJKLM 976
MABCDEFGHIJKLM 975
LABCDEFGHIJKLM 974
KABCDEFGHIJKLM 973
JABCDEFGHIJKLM 972
IABCDEFGHIJKLM 971
HABCDEFGHIJKLM 970
GABCDEFGHIJKLM 969
FABCDEFGHIJKLM 968
EABCDEFGHIJKLM 967
DABCDEFGHIJKLM 966
CABCDEFGHIJKLM 965
BABCDEFGHIJKLM 964
AABCDEFGHIJKLM 963
```

```
AABCDEFGHIJKLM 963
@ABCDEFGHIJKLM 962
```

```
^A 68
]A 67
\A 66
[A 65
ZA 64
YA 63
XA 62
WA 61
VA 60
UA 59
TA 58
SA 57
RA 56
QA 55
PA 54
OA 53
NA 52
MA 51
LA 50
KA 49
JA 48
IA 47
HA 46
GA 45
FA 44
EA 43
DA 42
CA 41
BA 40
AA 39
@A 38
```

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

```
===sort by file size===
EA 1024
qq 1024
pp 1024
oo 1024
nn 1024
mm 1024
ll 1024
kk 1024
jj 1024
ii 1024
hh 1024
```

```
gg 1024
ff 1024
ee 1024
dd 1024
cc 1024
bb 1024
aa 1024
~ABCDEFGHIJKLM 1024
}ABCDEFGHIJKLM 1023
|ABCDEFGHIJKLM 1022
{ABCDEFGHIJKLM 1021
zABCDEFGHIJKLM 1020
yABCDEFGHIJKLM 1019
xABCDEFGHIJKLM 1018
wABCDEFGHIJKLM 1017
vABCDEFGHIJKLM 1016
uABCDEFGHIJKLM 1015
tABCDEFGHIJKLM 1014
sABCDEFGHIJKLM 1013
rABCDEFGHIJKLM 1012
qABCDEFGHIJKLM 1011
pABCDEFGHIJKLM 1010
oABCDEFGHIJKLM 1009
nABCDEFGHIJKLM 1008
mABCDEFGHIJKLM 1007
lABCDEFGHIJKLM 1006
kABCDEFGHIJKLM 1005
jABCDEFGHIJKLM 1004
iABCDEFGHIJKLM 1003
hABCDEFGHIJKLM 1002
gABCDEFGHIJKLM 1001
fABCDEFGHIJKLM 1000
eABCDEFGHIJKLM 999
```

```
dABCDEFGHIJKLM 998
cABCDEFGHIJKLM 997
bABCDEFGHIJKLM 996
aABCDEFGHIJKLM 995
`ABCDEFGHIJKLM 994
_ABCDEFGHIJKLM 993
^ABCDEFGHIJKLM 992
```

```
^A 68
]A 67
\A 66
[A 65
ZA 64
YA 63
XA 62
WA 61
VA 60
UA 59
TA 58
SA 57
RA 56
QA 55
PA 54
OA 53
NA 52
MA 51
LA 50
KA 49
JA 48
IA 47
HA 46
GA 45
FA 44
DA 42
```

```
DA 42
CA 41
BA 40
AA 39
@A 38
?A 37
>A 36
=A 35
<A 34
;A 33
*ABCDEFGHIJKLMNOPQR 33
:A 32
)ABCDEFGHIJKLMNOPQR 32
9A 31
(ABCDEFGHIJKLMNOPQR 31
8A 30
'ABCDEFGHIJKLMNOPQR 30
7A 29
&ABCDEFGHIJKLMNOPQR 29
6A 28
5A 27
4A 26
3A 25
2A 24
b.txt 12
```

V. Learning

During this implementation, I get more familiar with the details of the file system strategy and knows some potential optimization points.