CSC3150 Assignment 4

CHEN Ang (*118010009*)

In this assignment, we are asked to implement a simple file system within the CUDA GPU kernel. The file system uses Global memory in the CUDA GPU as a volume (logical drive) and only contains one root directory. The information of the volume is directly saved in and retrieved from the volume (in Global memory), instead of being separately saved in Shared memory. Essential file system operations, including file opening, writing, reading, file removal and listing are supported.

Environment

OS: `Windows 10.0.19042 Build 19042`

CUDA (nvcc) version: `Release 10.2, V10.2.89`

GPU model: Nvidia GeForce GTX 970

How to Run the Program

The program was compiled by evoking `nvcc 10.2` in the terminal directly.

Under the `source` directory, enter the followings to compile a `main.exe` executable.

```
nvcc file_system.cu user_program.cu main.cu -dc
nvcc file_system.obj user_program.obj main.obj -o main
```

To run the executable, enter

```
./main.exe
```

Program Design

The majority of the design of the file system is in line with the specification given: The volume of the file system is decomposed into three parts summing up to `1060 KB` (with a file counter `nfiles` added)

```
struct FileSystem {                /* File System (Volume)        1060 KB  */
    u32   super[N_SUPERBLOCKS];    /* block occupancy             4    KB  */
    File  files[FCB_ENTRIES];      /* 1K FCB's                    32   KB  */
    uchar data[DATA_SIZE];         /* file contents in 32K blocks 1024 KB  */
    u32   nfiles;
};
```

The `super[N_SUPERBLOCKS == 1024]` bit-array stores the occupancy information of the `1 << 15` data blocks each of size `DATA_BLOCK_SIZE == 1024 KB`. The bit is defined to be `0` if the block is occupied (`1` if it is unoccupied) and is to be read from LSB to MSB as shown by the following examples:

```
/* All blocks initialized at 1 : unoccupied */
          |--------------super[0]---------------|  |super[1]|  |super[2]|
super : [1111 1111 1111 1111 1111 1111 1111 1111, 0xffffffff, 0xffffffff, ...]
block :  (31) .... .... .... .... ..98 7654 3210  (63)..(32)  (95)..(64)  ...

/* Block 5 of superblock 0 occupied! */
          |--------------super[0]---------------|  |super[1]|  |super[2]|
super : [1111 1111 1111 1111 1111 1111 1101 1111, 0xffffffff, 0xffffffff, ...]
block :  (31) .... .... .... .... ..98 7654 3210  (63)..(32)  (95)..(64)  ...
```

Notice that instead of a `1 << 12`-element `uchar` array, we change the design such that each of the superblock now stores occupancy information of 32 data blocks by declaring a `u32` array. This change from the template design is due to a `MAX_FILE_SIZE` of `1 KB`, which takes up exactly 32 data blocks. With a 32-block superblock design, a file can always fit into one empty superblock. This makes later implementation much more convenient.

`data` is a `uchar` array of `1024 KB` storing the actual content of the files. The formula of conversion between the index of a data block, i_B, the index (within `data`) of the first byte of that block, i_b, the index (within `super`) of the superblock the data block is in, i_S, and the block offset of the data block with respect to the first data block in the superblock, i_o, is given below:

$$
\begin{aligned}
i_b &= i_B \cdot 32 \\
i_S &= i_B / 32 \\
i_o &= i_B \bmod 32
\end{aligned}
$$

The `files[FCB_ENTRIES == 1024]` is an array of `File` structs, which are our implementation of the file control blocks. The `File` struct can be accessed directly through its index within the `files` array, which naturally becomes its file pointer or ID. Each `File` struct is of `32 bytes`, consisting of four data fields:

```
struct File {                        /* File Control Block          32 bytes */
    char fname[MAX_FILENAME_SIZE];   /* file name (null terminated) 20 bytes */
    u32  starting_block;             /* starting data block number  4  bytes */
    u32  fsize;                      /* file size in bytes          4  bytes */
    u32  btime;                      /* last modified time          4  bytes */
};
```

Since the allocation mechanism of this file system is contiguous, i.e., a file can only take up contiguous data blocks, to keep track of where the file is located we only need to store the index of the starting block, and the rest of the blocks can be deduced from the size of the file. The `starting_block` is set to that index if the file is present in the system (we say the `file` struct has been **activated**), and is set to `EMPTY == 0xffffffff` otherwise (`file` struct has been **deactivated**).

Before decomposing the implementation of `fs` operations, we need two helper functions:

- `u32 fs_find_empty_fp(FileSystem* fs)` scans linearly the `files` array and returns the `fp` of the first deactivated `file`. Returns `EMPTY` if all `file` structs have been activated.
- `u32 fs_find_name(FileSystem* fs, const char fname[])` scans linearly the `files` array and returns the `fp` of the file with name `fname`. (We assume the names are distinct.) Returns `EMPTY` if the name was not found. As a slight optimization, we keep track of the number of files scanned and break out the loop once `fs->nfiles` is reached.

We explain `fs_open`, `fs_read`, `fs_write`, and `fs_gsys` in more details below.

- `u32 fs_open(FileSystem* fs, const char fname[], int op)`

  To open a file, we first try to find `fname` in the file system using `fs_find_name`. If the name was found, we update the file's `btime` and return its `fp`. When `fs_find_name` failed to find such a name, we create a new file by first `fs_find_empty_fp` and activate the `file` struct found. Then we find an empty block by searching `1` bit in `super` (using some bit magic in `first_1bit`), flip that bit to `0`, and assign the block index to `starting_block` of the file. Finally, the `nfiles` counter is incremented by `1`, and the `fp` of the newly activated `file` returned.

- `void fs_read(FileSystem *fs, uchar* output, u32 size, u32 fp)`

  The implementation of `fs_read` is rather straightforward. We compute the starting byte of the data block from `starting_block` of the file, and then `memcpy` data of `size` bytes to `output`. The `btime` of the file is updated at last.

- `void fs_write(FileSystem *fs, uchar* input, u32 size, u32 fp)`

  `fs_write` splits into two cases based on whether or not the input content can fit into the original data blocks. If it can, then there's little work we need to do - simply `memcpy` the content of `input` to the original data blocks, update the occupancy bits in `super` as well as the file info accordingly. However, if the data blocks newly needed are larger than the original, we have to find some larger blocks that are unoccupied to write the content. To this end, we first mark the old blocks as unoccupied temporarily. Then we search for consecutive `1` bits in `super` which mark consecutive free blocks large enough to contain the new content. The searching is done with the help of `first_n_1bit` function, which returns the location of the first consecutive `1` bits of length `n` in a `u32` integer. If no empty blocks are large enough to contain the input, the function resumes the previously flipped bits in `super` and outputs an error. If found, these new blocks are marked occupied in `super`, and input content is copied to the new data blocks.

- `void fs_gsys(FileSystem* fs, int op)` (Files listing)

  The file listing operation is implemented by first fetching an array of `fp` of currently activated `file`'s, and then sorting the array in the way specified by `op` using Bubble Sort. The info of the `file`'s is retrieved from `fs` and printed in the order of the sorted array.

- `void fs_gsys(FileSystem* fs, int op, const char fname[])` (File removal)

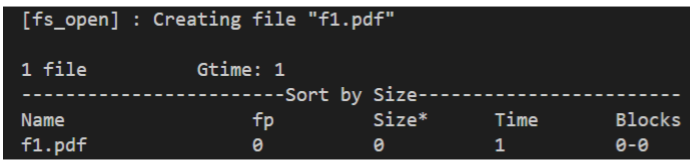
  To remove a file from the file system, simply flip the bits of occupied blocks in `super` and deactivate the `file` struct by setting the `starting_block` to be `EMPTY`. `nfiles` counter is decremented by `1`.

Problems I met

This project went smooth overall and most difficulties I met were minor, e.g., some typos in the code breaking the program which were fairly easy to debug.

Screenshots

- File creation



- File opening and writing

```
[fs_open] : File "f1.pdf" opened, fp: 0
[fs_write] : 10 bytes written to file "f1.pdf"!

1 file          Gtime: 3
-----------------------Sort by Size-----------------------
Name                    fp          Size*       Time        Blocks
f1.pdf                  0           10          3           0-0
```

- File reading

```
[fs_read] : 10 bytes of file "f1.pdf" read to output buffer.
```

- File listing (by size)

```
3 files         Gtime: 8
-----------------------Sort by Size-----------------------
Name                    fp          Size*       Time        Blocks
f2.exe                  1           1000        6           32-63
a3.tex                  2           100         8           1-4
f1.pdf                  0           10          4           0-0
```

- File listing (by time)

```
3 files         Gtime: 8
-----------------------Sort by Time-----------------------
Name                    fp          Size        Time*       Blocks
a3.tex                  2           100         8           1-4
f2.exe                  1           1000        6           32-63
f1.pdf                  0           10          4           0-0
```

- File listing (by file ID)

```
3 files         Gtime: 8
------------------------Sort by fp------------------------
Name                    fp*         Size        Time        Blocks
f1.pdf                  0           10          4           0-0
f2.exe                  1           1000        6           32-63
a3.tex                  2           100         8           1-4
```

- File removal

```
[fs_gsys] : File "a3.tex" removed!

2 files         Gtime: 8
------------------------Sort by fp------------------------
Name                    fp*         Size        Time        Blocks
f1.pdf                  0           10          4           0-0
f2.exe                  1           1000        6           32-63
```

- Output of **Test Case 1**

```
[fs_open] : Creating file "t.txt"
[fs_write] : 64 bytes written to file "t.txt"!
[fs_open] : Creating file "b.txt"
[fs_write] : 32 bytes written to file "b.txt"!
[fs_open] : File "t.txt" opened, fp: 0
[fs_write] : 32 bytes written to file "t.txt"!
[fs_open] : File "t.txt" opened, fp: 0
[fs_read] : 32 bytes of file "t.txt" read to output buffer.

2 files         Gtime: 8
-----------------------Sort by Time-----------------------
Name                    fp          Size        Time*       Blocks
t.txt                   0           32          8           0-0
b.txt                   1           32          4           2-2


2 files         Gtime: 8
-----------------------Sort by Size-----------------------
Name                    fp          Size*       Time        Blocks
t.txt                   0           32          8           0-0
b.txt                   1           32          4           2-2

[fs_open] : File "b.txt" opened, fp: 1
[fs_write] : 12 bytes written to file "b.txt"!

2 files         Gtime: 10
-----------------------Sort by Size-----------------------
Name                    fp          Size*       Time        Blocks
t.txt                   0           32          8           0-0
b.txt                   1           12          10          2-2


2 files         Gtime: 10
-----------------------Sort by Time-----------------------
Name                    fp          Size        Time*       Blocks
b.txt                   1           12          10          2-2
t.txt                   0           32          8           0-0

[fs_gsys] : File "t.txt" removed!

1 file          Gtime: 10
-----------------------Sort by Size-----------------------
Name                    fp          Size*       Time        Blocks
b.txt                   1           12          10          2-2
```

What I learned

I learned the basic working principles of a simple file system, particularly the contiguous allocation of memory blocks for files. I get to know how each file can be abstractly represented as a file control block and what actually happens underneath when one creates, opens, reads/writes and removes a file. I learned how a bit array can be used to compactly store booleans. Finally, I learned some cool functional programming ideas when writing codes for file sorting, namely implementing one bubble sort algorithm and then passing comparison lambda expressions as an argument of the algorithm to sort the array by any desired order!