File-System Interface Implementation

Yohandi [SID: 120040025]

Background

Data can be stored on a variety of computer-accessible media, including magnetic disks, magnetic tapes, and optical disks. To make the computer system user-friendly, the operating system presents a consistent logical picture of all data. To construct a logical storage unit, the file, the operating system must first abstract from the physical characteristics of its storage devices. The OS physically stores files on whatever the user specifies. Their data remains unchanged even after a system restart, as their nature is nonvolatile.

For all practical purposes, a file only exists in the abstract. Files are best defined by considering the many tasks that may be done on them. The OS makes system calls available to make it possible to make new files, read existing ones, move files around, delete files, and truncate existing ones. Here, we'll look at the OS requirements for executing these six fundamental file operations. Other related procedures, like renaming a file, should become intuitively obvious.

Program Design

In this task, we are asked to simulate six file management operations: open a file, write to a file, read from a file, list all available files listed by updated time, display all accessible files sorted by file size, and delete a file. In addition to those six activities, we also developed nine utilities to ensure they run well. We need to create an FCB design that will be the basis of our implementation if we want all of the features to perform as intended. All of our file metadata is kept in a database called FCB. Finally, after much deliberation, we settled on the following FCB layout.

For the FCB itself, we are free to use it however we like. Each FCB size is fixed to 32 bytes. Since there are at most 1024 files, we are expected to use a total of 1024×32 bytes from the `volume` array in the File System. For a file, we need to store the filename (which consists of 20 bytes), the time when it was created (which consists of 4 bytes = 256^4 possible values), the time when it was last modified (which consists of 4 bytes = 256^4 possible values), the size of the file (which consists of 2 bytes = 256^2 possible values), and the starting position for the file in the storage block (which consists of 2 bytes = 256^2 possible values). Adding those numbers, we have $20 + 4 + 4 + 2 + 2 = 32$, which is our exact number.

The six file management operations are included in the five main functions of the program. Those are: `fs_open()`, `fs_read()`, `fs_write()`, `fs_gsys()` for displaying the outputs, and `fs_gsys()` for deleting the file.

- `fs_open()` This procedure is in charge of performing the open operation on the file system and returning a pointer to the memory location of the opened file. The function first verifies the existence of the requested file name in the FCB section of the volume buffer. To save the entry, simply set the existing flag and update the record if the file name discovered matches the desired one. In such circumstances, there are two potential scenarios to think about: the current file already exists, and the current file does not exist. Two steps are necessary to create a file if the current file does not exist. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.

- `fs_read()` The contents of the requested address are simply copied into the output buffer, making read operations very straightforward. The size parameter's byte count determines how many items are taken. We use a system call to tell the computer which file to read and where in memory to store the data from the file as it is read. Once more, the directory is combed for the corresponding item, and a read pointer is maintained so the system always knows where it stands concerning the file. After a successful read, the read pointer is modified. Due to the binary nature of most processes' file operations, the current operation location may be stored as a current file-position pointer for each process. This single pointer is used for reading and writing, simplifying the system, and saving on storage.

- `fs_write()` In the first place, we have to make sure that any preexisting files have been removed. Therefore, we change the value of every element in the requested chunk to 0 at the very beginning of the function's source code (empty). Finally, we use the `fp` parameter to determine how the input is stored in memory (pointer from an open operation). The bitmap is then modified to 1. To save data to a file, we issue a system call and provide it the file's name and the data to save. The computer looks in the specified directory for a file with the given name. The system is responsible for maintaining a write pointer that points to the upcoming write position in the file. Every time a write is made, the write pointer must be incremented.

- `fs_gsys(LS_D / LS_S)` When the parameter is set to `LS_D`, the function is expected to list all file names in the directory and order by modified time of files. On the other hand, when the parameter is set to `LS_S`, the function is expected to list all file names and sizes in the directory and order by size. We first keep track of each file entry that exists. Once the correct entry is located in the directory, the pointer to the active file is advanced to the specified offset. Moving data inside a file can be done without requiring any I/O operations. A seek is another name for this action on a file. In addition, we can use a standard bubble sort algorithm (as it is easy to implement) to sort the file before displaying it to the user. The detail for the comparison is not necessary to be discussed; however, one vital key to be noted is that we need to involve a combination of the logic operations. The $O(N^2)$ bubble sort algorithm works pretty well because N, which denotes the number of files, is less than or equal to 1024. Of course, an algorithm such as merge sort exists and provides $O(NlogN)$ computation; however, it is only necessary when N is relatively large enough, such as $N \geq 10^5$.

- `fs_gsys(RM, filename)` When a file needs to be removed, it is first looked for in the directory structure. After locating the matching directory entry, we delete it and make the freed space available for use by other files. As the implementation, we first scan the FCB and compare it to the current configuration. Whenever a file with the specified name is retrieved, the corresponding index should be saved. After determining the index of the desired entry, we set the bitmap to 1 and replace all occurrences of 0 in the FCB and storage.

Program Execution

Program Environment

Note that to execute the program, it is already assumed that the user has already set up access to the school's computation resources.

Login to Cluster

- Open a new terminal and type `ssh {Student_ID}@CSC4005_cluster` to connect to the remote cluster login with SSH protocol.
- Enter a password to log in.

Transfer Files to Cluster

- Open a new terminal and locate the files that will be transferred.
- Type `scp {file} {Student_ID}@CSC4005_cluster:`. Here, a file can be in the format of `FILENAME.zip` if multiple files are to be sent. This will send files directly to `nfsmnt/{Student_ID}` destination.

Execution Steps

- After login in to the cluster, extract the zip files if the files are zipped.
- It is required that in the destination, there are exactly 6 files: `data.bin`, `main.cu`, `slurm.sh`, `user_program.cu`, `file_system.cu`, and `file_system.h`.
- To submit a batch, type `sbatch ./slurm.sh`. After execution is done, new files named `result.out` and `snapshot.bin` will be printed in the same destination. `result.out` shows the compile and run result.
- Another method is by typing both `nvcc --relocatable-device-code=true main.cu user_program.cu virtual_memory.cu -o test` to directly compile the CUDA script using the nvcc compiler and `srun ./test` to run the compiled execution file. Note that this step produces the same

result as the previous step; however, `result.out` will not be printed as the content will be printed directly to the terminal.

Transfer Files from Cluster

- If the current terminal is still in the cluster root, type `exit` to exit from the cluster.
- Type `scp 120040025@CSC4005_cluster:~/{file} .` to retrieve the file back to the current folder.

Result

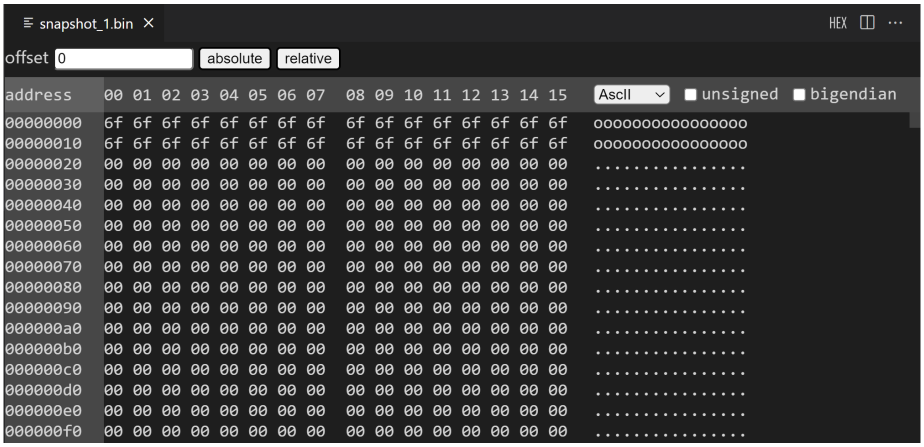
When changing a test case, kindly edit the `user_program.cu` file and transfer it back to the cluster.

Test Case #1

```
/////////////// Test Case 1  ///////////////
u32 fp = fs_open(fs, "t.txt\0", G_WRITE);
fs_write(fs, input, 64, fp);
fp = fs_open(fs, "b.txt\0", G_WRITE);
fs_write(fs, input + 32, 32, fp);
fp = fs_open(fs, "t.txt\0", G_WRITE);
fs_write(fs, input + 32, 32, fp);
fp = fs_open(fs, "t.txt\0", G_READ);
fs_read(fs, output, 32, fp);
fs_gsys(fs,LS_D);
fs_gsys(fs, LS_S);
fp = fs_open(fs, "b.txt\0", G_WRITE);
fs_write(fs, input + 64, 12, fp);
fs_gsys(fs, LS_S);
fs_gsys(fs, LS_D);
fs_gsys(fs, RM, "t.txt\0");
fs_gsys(fs, LS_S);
```

In this case, we first create a file named `t.txt` and write content with a size of 64 bytes from the first 64 bytes in `data.bin`. Then, we create another file named `b.txt` and write content with a size of 32 bytes from [32..63] in `data.bin`. We do the exact same previous operation for `t.txt`.

After the first 3 writing cases, the program will then read the value from the first 32 bytes in `t.txt` and write it to `snapshot.bin`. This is the reason why `snapshot.bin` is filled with 32 bytes from [32..63] in `data.bin`.



snapshot.bin

After the reading case, the program will then list the existing files sorted by two states: `LS_D` and `LS_S`. With a `LS_D` state, it will list the files sorted by their last modified time. This will print `t.txt` then `b.txt` since `t.txt` was the last modified version. With a `LS_S` state, it will list the files sorted by their file size. This will print `t.txt 32` and then `b.txt 32` since `t.txt` was the first one to be created.

The program will then write the content of 12 bytes from [64..75] in `data.bin`.

After the writing case, the program will again list the existing files sorted by two states: `LS_D` and `LS_S`. With a `LS_S` state, it will list the files sorted by their file size. This will print `t.txt 32` then `b.txt 12` since `t.txt` has a bigger size than `b.txt`. With a `LS_D` state, it will list the files sorted by their last modified time. This will print `b.txt` then `t.txt` since `b.txt` was the last modified version.

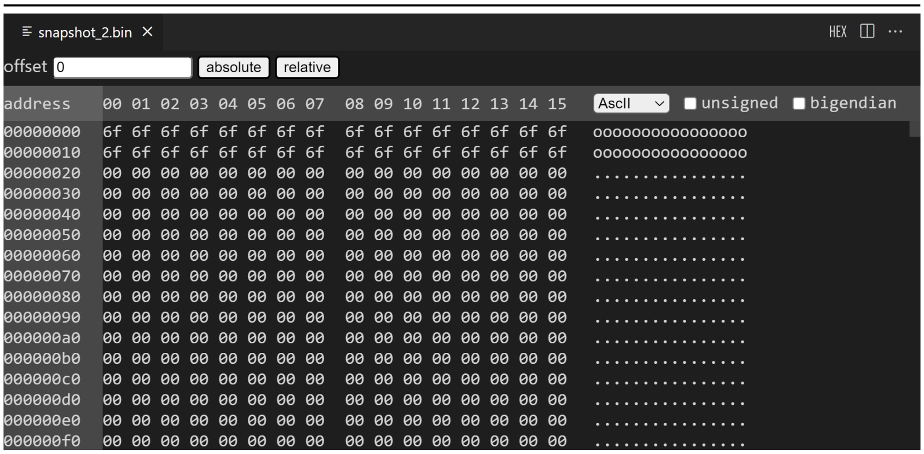
The program will remove the `t.txt` file and list the existing files with `LS_S` state. This will print `b.txt 12` since `b.txt` is the only file.

```
===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
```

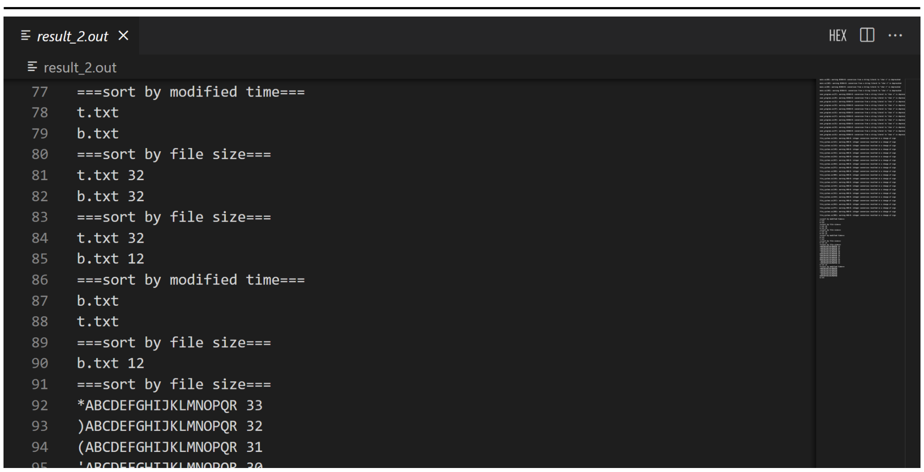
result.out

Test Case #2

```
/////////////// Test Case 2  ///////////////
u32 fp = fs_open(fs, "t.txt\0", G_WRITE);
fs_write(fs,input, 64, fp);
fp = fs_open(fs,"b.txt\0", G_WRITE);
fs_write(fs,input + 32, 32, fp);
fp = fs_open(fs,"t.txt\0", G_WRITE);
fs_write(fs,input + 32, 32, fp);
fp = fs_open(fs,"t.txt\0", G_READ);
fs_read(fs,output, 32, fp);
fs_gsys(fs,LS_D);
fs_gsys(fs,LS_S);
fp = fs_open(fs,"b.txt\0", G_WRITE);
fs_write(fs,input + 64, 12, fp);
fs_gsys(fs,LS_S);
fs_gsys(fs,LS_D);
fs_gsys(fs,RM, "t.txt\0");
...
```



snapshot.bin



result.out

Test Case #3

```
/////////////// Test Case 3  ///////////////
u32 fp = fs_open(fs, "t.txt\0", G_WRITE);
fs_write(fs, input, 64, fp);
fp = fs_open(fs, "b.txt\0", G_WRITE);
fs_write(fs, input + 32, 32, fp);
fp = fs_open(fs, "t.txt\0", G_WRITE);
fs_write(fs, input + 32, 32, fp);
fp = fs_open(fs, "t.txt\0", G_READ);
fs_read(fs, output, 32, fp);
fs_gsys(fs, LS_D);
fs_gsys(fs, LS_S);
fp = fs_open(fs, "b.txt\0", G_WRITE);
fs_write(fs, input + 64, 12, fp);
fs_gsys(fs, LS_S);
fs_gsys(fs, LS_D);
fs_gsys(fs, RM, "t.txt\0");
...
```

```
snapshot_3.bin                                                          HEX
offset 0   absolute  relative
address   00 01 02 03 04 05 06 07  08 09 10 11 12 13 14 15   AscII  unsigned  bigendian
00000000  6f 6f 6f 6f 6f 6f 6f 6f  6f 6f 6f 6f 6f 6f 6f 6f   oooooooooooooooo
00000010  6f 6f 6f 6f 6f 6f 6f 6f  6f 6f 6f 6f 6f 6f 6f 6f   oooooooooooooooo
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   ................
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000000e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000000f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
```

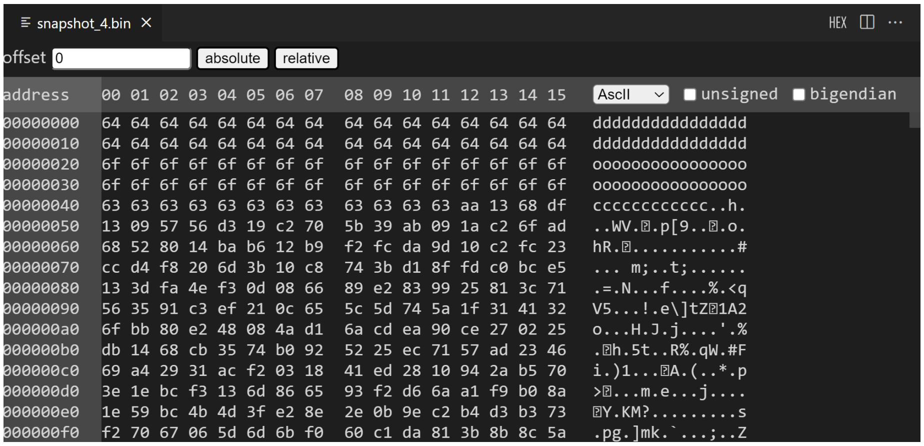
snapshot.bin

```
===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
```

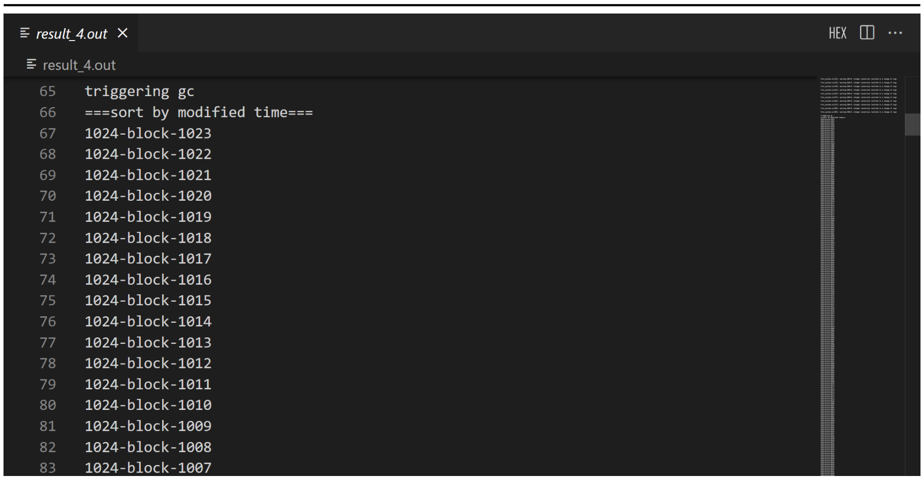
result.out

Test Case #4

```
/////////////// Test Case 4  ///////////////
u32 fp = fs_open(fs, "32-block-0", G_WRITE);
fs_write(fs, input, 32, fp);
for (int j = 0; j < 1023; ++j) {
    char tag[] = "1024-block-????";
    int i = j;
    tag[11] = static_cast<char>(i / 1000 + '0');
    i = i % 1000;
    tag[12] = static_cast<char>(i / 100 + '0');
    i = i % 100;
    tag[13] = static_cast<char>(i / 10 + '0');
    i = i % 10;
    tag[14] = static_cast<char>(i + '0');
    fp = fs_open(fs, tag, G_WRITE);
    fs_write(fs, input + j * 1024, 1024, fp);
}
...
```



First 16 lines of snapshot.bin



result.out

Reflection

The basics of CUDA programming were our first lesson in this task. Conversely, we picked up the bitmap technique for documenting the files already in memory and the free space between them. Finally, we have a general idea of how to implement the sorting mechanism in the file system when showing the existing files and what happens when the remove operation is asked to delete a file. Considering the significance of the file system handled by the operating system, this project invites students to play the role of the operating system in carrying out certain activities. Finally, it has been demonstrated that the program runs all test cases successfully.