Project 3: Report

BY 冯璇 520021911147

Table of contents

1 Step 1

1.1 Basic Idea & Code Logic

The requirement of this part is to implement a simulation of physical disk, which works on mainly 4 kinds of requests: I(get information), R(read certain block), W(write certain block), E(exit).

First of all, the disk is simulated with the structure of cylinders and sectors. The number of cylinders and the number of sectors per cylinder are assigned by user, while the sector size is fixed to 256B. The program create a file to simulate physical disk, and map a block of memory space to the file with mmap() to make read and write operations faster.

The program code logic is as following, for full version of the codes please check `/src/step1/disk.c`;

- The program first open the file that will act as the disk, and use lseek() to stretch it to the required size (which is `CYLINDERS * SECTORS_PER_CYLINDER * SECTOR_SIZE`); then tyr to write an empty string to the end of the file in order to make sure that the stretching process successes;
- Next the program use mmap() to map a block of memory space to the disk file;
- After all the preperation has been successfully done, the program will get lines of instructions from STDIN and parse the arguments;
- For "I" request, the program will print the arguments of the disk, including number of cylinders, number of sectors per cylinder;

- For “R” request, the program jumps to the required sector of the required cylinder, simulating the track delay time with usleep(); then the program uses memcpy() to get the content of the block, and prints it to both out_log and STDOUT;

- For “W” request, the program jumps to the required sector of the required cylinder, simulating the track delay time with usleep(); then the program uses memcpy() to copy the content of data_buffer to the block, and prints “Yes” to both out_log and STDOUT;

1.2 Result Demonstration

Compile and run `./disk`, input the following requests to find the results as shown in Fig. 1:

```
(base) yvesfung@yvesfung:~/local/proj3/src/step1$ make
gcc -Wall disk.c -o disk
(base) yvesfung@yvesfung:~/local/proj3/src/step1$ ./disk 5 12 10 diskFile
==>I
#cylinders: 5, #sectors per cylinder: 12
==>W 4 7 helloWorld!
Yes
==>R 4 7
Yes helloWorld!

==>W 8 9 helloWorld!
No
==>R 6 10
No
==>W 4 7 how are you?
Yes
==>R 4 7
Yes how are you?

==>e
Goodbye!
```

Fig. 1: Command line output of disk

And the content of disk.log is as shown in Fig. 2:

```
(base) yvesfung@yvesfung:~/local/proj3/src/step1$ cat disk.log
#cylinders: 5, #sectors per cylinder: 12
Yes
Yes helloWorld!

No
No
Yes
Yes how are you?

Goodbye!
```

Fig. 2: Content of disk.log

We can see that the both the output of command line and the output of disk.log are correspondent with our expectation.

2 Step 2

2.1 Basic Idea & Code Logic

The requirement of this part is to implement a simulation of a file system. The structure I implemented is FAT.

The structures created for this file system simulation includ boot_block, fcb, fat, and user_open, of which the details could be checked in fs.h; and for all the function prototypes please also check fs.h.

2.1.1 System Structure

Block0 of the system is the boot block, block1 and block2 are for fat0, block3 and block4 are for fat1, block 5 and block 6 are for root directory, and the data blocks start from block7;

The program first allocates memory space for the filesystem. After each time the system is closed, the current state of the system will be stored in a file "fsFile". And next time when the system is started, it will first try to open "fsFile", by which last time state could be restored; if no such file exists, the system will create "fsFile" and format it;

2.1.2 Functions Logic Explanation

- **int start_sys(void)**

  to start the system;

  The function allocates memory space and open fsFile, then opens and initializes root directory, and then initializes the global variables;

- **int Format(void)**

  to format the system;

  The function sets the boot block, and allocates fat0's and fat1's blocks, then creates root directory and allocates blocks for the root's "." and ".." directories;

- **int Chdir(char **args)**

  to change directory;

  The function first check is the argument is legal, then check if the directory required to is open, if not then open it, and finally set the directory as current directory;

- **int Pwd(void)**

  to print current directory;

  The function prints out current directory's name;

- **int Mkdir(char **args)**

- **int in_mkdir(const char *parent_path, const char *dirname)**

  to make directory;

  These two functions together finish the work of making a directory: Mkdir() will check the arguments first and get the parent_path and dir_name, and call in_mkdir;

in_mkdir() will check for free fcb and free disk space, then set a new fcb and allocate "." and ".." directories for the new directory;

- **int Rmdir(char **args)**
- **void in_rmdir(fcb *dir)**

  to remove directory;

  These two functions together finish the work of removing a directory: Rmdir() will check the arguments and find the directory's fcb, then call in_rmdir();

  rmdir() will set "." and ".." directories free, reclaim the disk space and finally modify FAT;

- **int Ls(char **args)**
- **void in_ls(int first, char mode)**

  to list directory, -l for more detailed information;

  prints folder name in green and file name in default color;

  Ls() gets the path and in_ls() list the fcb's names under this path;

- **int Create(char **args)**
- **int in_create(const char *parpath, const char *filename)**

  to create file;

  Create() will parse the arguments and check if they're legal, for example, if the file already exists, and then call in_create;

  in_create() will check if there's free fcb under current directory and if FAT has free blocks, then separate the new file name's fileName and extensionName, and finally set a new fcb for the file.

- **int Rm(char **args)**

  to remove file;

  Rm() will first check if the file exists, then if the file is open we need to close it first before removing; finally the function reclaims the disk blocks of the file and modifies FAT;

- **int Write(char **args)**
- **int do_write(int fd, char *content, size_t len, int wstyle)**

  to write a file, -a for add, -i for insert, -w for write, user needs to double tap ENTER to end input;

  Write() parses the arguments and sets the writing mode(-a/-i/-w), according to which find the block index of the file to be written; then prints out the prompt to get user input, and finally calls do_write();

  do_write() first prunes and concatenates the user input data according to the requirements of different modes, and then organizes it into new file content; after this, the function will write the data to the disk in blocks. If the original number of blocks is not enough, new blocks will be applied for the file. If the original number of blocks is redundant, the excess blocks will be released.

- **int Del(char **args)**

  to delete characters from certain character in a file;

Del() will parse the arguments and then call do_write, to write '\0' to the required position;

- **int Read(char **args)**

- **int do_read(int fd, int len, char *text)**

  to read a file and print;

  Read() first checks if the arguments are legal, for example, if the input filename is a folder's name, if the file exists, if the file is open; then finds the file's first block in disk and calls do_read();

  do_read() will first check if the lenth to be read covers the end of the file, then get the data into data_buffer to print out;

- **int exit_sys()**

  to exit the system;

  This function closes all open files and stores the content of memory space into fsFile, then closes fsFile and free memory space and virtual disk space;

2.2 Result Demonstration

First test **f(format), mkdir, rmdir, cd, ls, pwd, exit**, as shown in Fig. 3:

```
(base) yvesfung@yvesfung:~/local/proj3/src/step2$ ./main
System initializing...
Format Done.
Initialized successfully
/ $ mkdir local1
/ $ ls
.       ..      local1
/ $ mkdir local2
/ $ ls -l
0           5     512  2023-05-26      21:32:42        .
0           5     512  2023-05-26      21:32:42        ..
0           7     256  2023-05-26      21:33:02        local1
0           8     256  2023-05-26      21:33:10        local2

/ $ cd local1
/local1 $ pwd
/local1
/local1 $ cd /local2
/local2 $ cd /
/ $ rmdir local1
/ $ ls
.       ..      local2
/ $ f
Format Done.
/ $ ls
.       ..
/ $ exit
```

Fig. 3: test f(format), mkdir, rmdir, pwd

Then test **mk, rm, ls**, as shown in Fig. 4:

```
(base) yvesfung@yvesfung:~/local/proj3/src/step2$ ./main
/ $ mkdir local
/ $ ls
.       ..      local
/ $ cd local
/local $ mk hello.c
/local $ ls
.       ..      hello.c
/local $ rm hello.c
/local $ ls
.       ..
/local $ mkdir new
/local $ ls
.       ..      new
/local $ cd new
/local/new $ mk new_file
/local/new $ ls
.       ..      new_file.d
/local/new $ rm new_file
rm: no such file
/local/new $ rm new_file.d
/local/new $ exit
```

Fig. 4: test mk, rm, ls

And finally test cat(read), write, del(delete content), as shown in Fig. 5:

```
(base) yvesfung@yvesfung:~/local/proj3/src/step2$ ./main
/ $ mk exp.py
/ $ write exp.py -w
print("hello")

/ $ cat exp.py
print("hello")
/ $ write exp.py -i
Please input location: 1
-you-

/ $ cat exp.py
p-you-rint("hello")
/ $ write exp.py -a
print("hi")

/ $ cat exp.py
p-you-rint("hello")
print("hi")
/ $ del exp.py
please input location: 19
/ $ cat exp.py
p-you-rint("hello")
/ $ exit
```

Fig. 5: test read, write, del

We can see that all of the output are correspondent with our expectation.

3 Step 3

3.1 Basic Idea & Code Logic

Since I've already used a file to simulate the disk in step 2, I did not add socket communication between the disk and the file system in view of time constraints, but I added the simulation of disk head movement delay in the file system's program;

Since what I implemented in step 2 was a shell-type program that could accept user requests, I did not implement an additional client, since it was the same as what I did in step 2.

4 Summary

In this project, I implemented a physical disk simulation and a file system simulation, and further added some functions of disk simulation to file system simulation to make it closer to the reality case in terms of the head movement and track delay time. From getting familiar to FAT system to finishing the whole project, I spent nearly 2 weeks learning and coding, which was indeed quite a lot work for me, but meawhile I did enjoy the process and gained a lot more than I thought.

Here are a **few things that could be improved further** to this project, and I will continue to try to complete these after the final examination weeks:

- add more error handling to make the program stronger;
- add mmap() in step 3 to make read/write faster;
- use execvp() to make the program support more request;
- add access control for certain files or folders;
- .....

Last but not least, I want to say thank you to Prof. Qinya Li, and also to the teaching assistants, who have really helped me a lot :)