Assignment #4: Basic File System Implementation

Due: April 9, 2021 at 23:55 on myCourses

1. Preliminary Notes

- The question is presented from a Linux point of view using the computer science server `mimi.cs.mcgill.ca`, which you can reach remotely using `ssh` or `putty` from your laptop (see lab 1). We suggest you do this assignment directly from `mimi.cs.mcgill.ca` since these are the computers the TA will be using.
- Your solution must be built using the modular programming techniques we discussed in the in-class exercise sessions and the C labs.
- **You must write this assignment in the C Programming language.**

2. Assignment Question: Building a Simple File System

2.1 Overview:

For this assignment, you will need the **fully working kernel and memory manager from Assignment #3**. You can either build on top of your past submission, or use the official solution provided on myCourses. The goal of this assignment is to add basic file system functionality to your kernel. This is a **contiguous** file system. For an **extra 4 bonus points,** you can make it not contiguous.

In this assignment we will look at three main components of a file system:

- **Higher-level kernel code** that provides an API for users to access the file system.
- **Driver(s)** that instruct the machine how to read and write data from/to the disk.
- **On-disk data structures** that are created to represent abstract concepts like directories and files. In this assignment, we will not implement the main memory file system data structures. These data structures are manipulated through the disk driver.

2.2 API:

You will create three additional commands for your scripting language. Since we will not implement the full file system functionality, our three commands will be limited to:

- `mount partitionName number_of_blocks block_size`
  The command mount uses the `partition()` and `mountFS()` functions, described below. If a partition already exists with the name provided then the existing partition is mounted, otherwise the provided arguments are used to create and mount the new partition. **All the arguments are required.**

- `write filename [a collection of words]`
  The `write` command checks to see if `filename` is open. If it is already open, then the words collection provided between brackets is written to the file (as defined in the driver). Otherwise, `write` first opens the file and then writes to it. The collection of words is a series of alphanumeric character strings separated by spaces with the open square bracket preceding the words and the close square bracket terminating the output. You do not need to support any other kind of formatting for the full

points, but you will not be penalized if you do.

- `read filename variable`
  The `read` command first checks if `filename` is open. If it is already open, then it reads from the file (as defined in the driver), otherwise it opens the file and then reads from it. The string that is returned is assigned to `variable` via the `set` command that is already implemented in your shell.

We now describe the data-structures, driver, and additional OS code you will need to implement for this assignment. Note that you will not need to remove/replace any major piece of code from the past assignments.

2.3 Disk Driver

The disk driver functionality is implemented in `DISK_driver.c` (and `DISK_driver.h`). Your driver will contain the following functions. More implementation details about each function are given in Section 2.4.

- `void initIO()`
  // initialize all global data structure and variables to zero or null. Called from your `boot()` function.
- `int partition(char *name, int blocksize, int totalblocks)`
  //create & format partition. Called from your `mount()` function that lives in the interpreter, associated to your scripting `mount` command.
- `int mountFS(char *name)`
  // load FAT & create buffer_block. Called from your `mount()` function that lives in the interpreter, associated to your scripting `mount` command.
- `int openfile(char *name)`
  //find filename or creates file if it does not exist, returns file's FAT index. Called from your scripting `read` and `write` commands in the interpreter.
- `char *read Block(int file)`
  // using the file FAT index number, load buffer with data from `current_location.` Return block data as string from `block_buffer.`
- `int writeBlock(int file, char *data)`
  // using the file FAT index number, write data to disk at `current_location`

2.4. File System Data Structures:

This section describes all the private data structures that need to be added to DISK_driver, and provides implementation details about the disk driver functions.

2.4.1 The Partition Table

```
struct PARTITION {
    int total_blocks;
    int block_size;
} partition;
```

The PARTITION structure records information about the format of the partition. A disk partition is composed of two areas: the partition header and the partition data area. The partition header contains the PARTITION structure and the File Allocation Table (FAT) structure (described next – read carefully, since this version of the FAT structure is different and simpler than what we have seen in class).

When a partition is created, information about the format of the partition is recorded at the beginning of the partition, followed by the directory tree (FAT), and finally the largest section is the partition data area where all the files are stored. In our example, `total_blocks` records the number of blocks available in the partition data area. The integer `block_size` records the byte size of a block. For example, if `total_blocks` is 10 and `block_size` is 5, then the total number of bytes in the partition data region is 50 bytes (divided into 10 blocks). **A byte will be represented as a character in our simulation.** The structures PARTITION and FAT are fixed size.

On disk, the format looks as follows, PARTITION : FAT : BLOCKS, where the colon means concatenation, and the word BLOCKS refers to the "partition data region". The data stored in your partition is persistent. It is there every time you rerun your assignment (and for your TA to see).

PARTITION is a private struct but global within DISK_driver.c. The PARTITION structure is created and accessed via the `partition()` and `mount()` functions.

`int partition(char *name, int blocksize, int totalblocks)`

- All other functions depend on the existence of a partition, so this is the first thing that needs to be called in `mount()`.
- To simulate a partition, we will use a subdirectory. `partition()` creates a directory called PARTITION (in caps) `(mkdir PARTITION)` if one does not exist in the current directory. Within PARTITION create a file having the `name` argument as its filename. Each partition is simulated by one file. The simulated files within the partition are stored as the contents of the partition file.
- Format the file as follows: the beginning of the file contains the information from `struct partition`, and then the information from `fat[20]` (initially empty, initialized thorough the `initIO()` function; see Section 2.3). It then appends the partition data area by writing `total_blocks * block_size` number of '0' (character zero) to the end of the file.
- If it is successful, then it returns 1 otherwise 0.

`int mountFS(char *name)`

- `mountFS()` uses the `name` argument as the partition filename. It opens the partition from the PARTITION directory and loads the information from the partition into the global structures `partition` and `fat[]`.
- This function will also malloc `block_size` bytes and assign that to `block_buffer` (see Section 2.4.3).
- If it is successful, then it returns 1 otherwise 0.

2.4.2 The File Allocation Table (FAT)

```
struct FAT {
    char *filename;
    int file_length;
    int blockPtrs[10];
    int current_location;
} fat[20];
```

The FAT contains information about all the files in a partition. It can store a **maximum of 20 files**. FAT is a private structure but global within DISK_driver.c. FAT contains:

- The field `filename` is the name of a file in the partition.
- The field `file_length` is the length of the **file in number of blocks**. Each block is equal to `block_size` bytes (char), defined in Section 2.4.1.
- The array `blockPtrs[]` are the pointers to the blocks in the partition populated with data from this file. These pointers are block numbers.
- The field `current_location` is a pointer to the current read/write location of the file in block number. It is initialized to -1 when not used.

2.4.3 The Block Buffer

```
char * block_buffer;
```

The `block_buffer` is malloced to the correct size when the `partition()` function is called. The `block_size` parameter is used in the malloc operation. The block buffer is used by the `read_block()` function to bring in blocks from disk. This is a private structure but global within DISK_driver.c.

2.4.4 The Active File Table

```
FILE *active_file_table[5];
```

The active file table contains all the system wide open file pointers, which is 5 maximum. This is a private structure but global within DISK_driver.c.

The FAT, Block Buffer and Active File Table are involved in the following functions:

`int openfile(char *name)`

- The `openfile()` function assumes that FAT contains data. It searches for the string `name` argument in the `filename` fields of `fat[20].`
- If it finds the file, then an available cell in the active file table is made to point to the first block of the file and the index of the FAT cell is returned.
- Otherwise, it creates a new entry in the FAT table, leaves the corresponding active file table cell NULL) and returns the FAT index to that new entry.
- To set an entry in the active file table you will need to `fopen()` the partition and then `fseek()` to the block.
- **You will also need another data structure to remember which `active_file_table` entries belong to which `fat[].`**
- If there are no available cells in the active file table, or there is no more space in the partition data area, or the FAT is full, or some other issue happened, it returns the value -1 to indicate an error occurred.

`int readBlock(int file)` and `int writeBlock(int file, char *data)`

- These functions need the FAT index number returned from `openfile()` as the input parameter. These functions fail when an invalid index number is given. The index number behaves like a pointer to the file. It tells the function where in the file we are currently at, through the `current_location` parameter in the FAT entry and which file pointer to use, through the

Active File Table entry. These functions are agnostic to the run-time situation. They perform no system wide checking. They handle only immediate read and write operations.

- In the case of `readBlock()`, the `current_location` is used to load the `block_buffer` with the data from the block, if not at end of file. It then increments the `current_location` integer and returns success. If at end of file, then `block_buffer` does not change, and the function returns failure.
- In the case of `writeBlock()` the `current_location` is used to write the data argument into the partition. This is a destructive block write**, not a destructive file write**. This means when a file has many blocks and we write to a particular block, the data in that block is overwritten but the rest of the file is not affected by that write. The FAT is updated correctly, and `current_location` is incremented to the next cell.

Note 1: It is important to note that `blockPtrs[]` points to the actual blocks in the partition file where the file data is stored. It is also important to note that `current_location` is used as an index for `blockPtrs[]`.

Note 2: You know when a block is not free because it is no longer initialized to '0'. Since we do not have a delete command, this property always holds.

2.5 Program termination

Program termination is like in assignment 3. **Remember that the partitions are persistent.** When your program terminates the TA will be able to see the files.

2.6 Testing your kernel

The TAs will use and modify the provided test file to test your kernel. Your executable should be called `mykernel`. To use the provided test file, you will need to run your program from the OS command line as follows: `$ ./mykernel < testfile.txt`

Make sure your program works in the above way.

WHAT TO HAND IN

Your assignment has a **due date on April 9, 2021 at 23:55** plus **two** late days. If you choose to submit your assignment during the late days, then your grade will be reduced by -5% per day. Submit your assignment to the Assignment #4 submission box in myCourses. You need to submit the following:

- Make sure to ZIP your assignment submission into a single file called ass4.zip
- A README.TXT file stating and any special instructions or comments you think the TA needs to know to run and grade your program.
- **Your version** of TESTFILE.TXT. This will be you telling the TA that you know for sure that your program can at least do the following. The TA will run your program with this file and they will also run it with their own version of the file. Feel free to modify the testfiles we provide.
- Submit all the .c file described in the assignment (you may want to create .h files, if so, please hand those in as well)
- Submit the executable (compiled on mimi).
- Submit a bash file with the gcc command to compile your program for mimi, or a Makefile.

Note: You must submit your own work. You can speak to each other for help but copied code will be handled as to McGill regulations. Submissions are automatically checked via plagiarism detection tools.

HOW IT WILL BE GRADED

Your assignment is graded out of **26 points (plus 4 bonus points)** and it will follow this rubric:

- The student is responsible to provide a working solution for every requirement.
- The TA grades each requirement proportionally. This means, if the requirement is only 40% correct (for instance), then the student receives only 40% of the points assigned to that requirement.
- **Your program must run to be graded.** If it does not run, then the student receives zero for the entire assignment. If your program only runs partially or sometimes, you should still hand it in. You will receive partial points.
- The TA looks at your source code only if the program runs (correctly or not). The TA looks at your code to verify that you (A) implemented the requirement as requested, and (B) to check if the submission was copied.

  Mark breakdown:
  - **1 point** – Source file names. The TA must verify that the student created the specified source files as the only source files in the application. In addition, the source files must be populated with at least what was specified in the assignment description for each file. The student is permitted to supply additional helper functions as they see fit, if it does not hinder the assignment requirements.
  - **1 point** – Modular programming. The assignment uses modular programming techniques.
  - **1 point** – The student's compiled program must be named `mykernel`.
  - **2 points** – A fully working Assignment 4.
  - **10 points** – DISK_driver.c – 1 point for each function and data structure/global variable
  - **3 points** – The mount command, including mountFS.
  - **3 points** – The read command
  - **3 points** – The write command
  - **2 points** – Program can be tested with the TESTFILE.TXT file
  - **Programming expectations.** As software students at the 300 and 400 level at McGill University, it is expected that you code at the level of a beginner professional software engineer, even when a programming assignment does not expressly state it. Your code needs to meet the programming style posted on myCourses. **If not, your TA may remove up to 5 points, as they see fit.**
  - **BONUS POINTS**
    - **4 points** for non-contiguous file system
      (add that to your README file so your TA notices)

Good luck and enjoy! At the end of this assignment, you will have a full OS prototype :)