COMP 304 Project 3

Mini File System
Corresponding TA: Mandana BagheriMarzijarani
Due: May 27, 2022

Notes: The project can be done **individually or in teams of 2**. You may discuss the problems with other teams and post questions to the OS discussion forum but the submitted work must be your own. This assignment is worth *10% of your overall grade*. It is strongly recommended to start working on the project as soon as possible.

Any material used from other sources, including the web, should be properly cited in the code and supporting files. Cheating will not be tolerated.

1 Description

In this project, you will implement a file system API library which operates on virtual disks. The file system is based on File Allocation Table (FAT), and it does not support directories, i.e., all files are stored flat in the virtual disk. The virtual disk which you will implement will be itself stored as a single *real* file, i.e., it will become a single file in your project directory when you run your code.

The file which functions as the virtual disk will have the following specifications:

- The virtual disk is divided into multiple fixed-sized regions called blocks.
- The number of blocks in the virtual disk are set (and fixed) at the time of its creation.
- The size of blocks (`BLOCK_SIZE`) is defined at the time of creation (e.g., 1024 bytes).
- A block is associated at most to a single file, so multiple files cannot share a block.
- The first block is the master record which contains the File Allocation Table.
- Each file in the virtual disk takes at least one block.

There are many ways of implementing the first block, but for the sake of simplicity, we will only store 3 items in it:

1. The number of blocks in the virtual disk (`BLOCK_COUNT`) as an integer.
2. The size of each block in the virtual disk (`BLOCK_SIZE`) as an integer.
3. `BLOCK_COUNT` bytes, each representing the type of the corresponding block in the virtual disk (`BLOCK_MAP`) as a number.

There are 4 block types as defined in `fat.h`. `EMPTY_BLOCK` denotes whether a block is used or still empty. `FILE_ENTRY_BLOCK` denotes the start of a file (includes file metadata but no file data). `FILE_DATA_BLOCK` denotes a block that contains file data. `MASTER_BLOCK` which is the first block of the virtual disk. You can see an illustration of the virtual disk with above mentioned specification in Figure 1.

You will need to implement two categories of functions. The first category are Disk Manipulation functions, which are named like `minifs_fat_*`. The second category are File Manipulation functions, which are named like `minifs_file_*`.

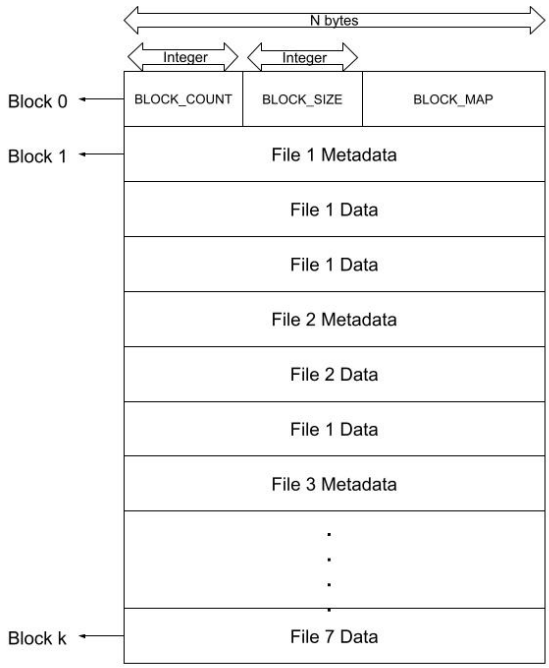

Figure 1: Virtual disk architecture

2 Disk Manipulation

The public API for disk manipulation includes the following functions as defined in `fat.h`:

1. `mini_fat_create(filename, block_size, block_count)` creates a new virtual disk and stores it inside the filename.

   The created virtual disk file should be exactly `block_count*block_size` bytes and should reside in your project directory immediately after creation.

   The function should also return a `FAT_FILESYSTEM` object with its fields set properly.

2. `mini_fat_save(fs)` which completely saves a file-system to the real file on the disk. After the save operation, the virtual file-system should be completely stored inside the single file passed to `create`, and can be loaded later.

   Note that the actual file data of the file-system (i.e., the `FILE_DATA_BLOCK` blocks) should not be stored in memory or in the file-system structs, meaning that the save API should at most write the file-system metadata and file metadata to the virtual disk file (if they are not already stored in other steps).

   The individual file data blocks should be stored/retrieved using file manipulation APIs as described later.

3. `mini_fat_load(filename)` should load a previously saved virtual disk file-system from a real file in your project directory. The loaded virtual file-system should be identical to the one that was saved, and all the same tests should pass on it.

4. `mini_fat_dump(fs)` which should dump the metadata of the virtual file-system and the virtual files within it, sans the actual file data.

There is no specific format for the dump, but it should be in a human readable format, and include all the information used to create the file-system and the files inside of it.

There are also multiple helper functions defined in `fat.h`, for example for finding an empty block in the file-system, allocating an empty block, writing data inside a block and reading data inside a block. You do not necessarily need to implement these functions, but doing so may help you have a more organized and well-structured code-base, as these helpers can be called from the File System Manipulation APIs as an abstraction.

Note that you only need to expose the public API in a way that satisfies the test suite provided in `main.cpp`, and do not need to implement the helper functions or any other function that is not part of the public API.

Also, you are free to change the innards of the structs used by these functions, as the structs are not directly accessed by user code, and are only passed around to API functions. As long as the API signature remains unchanged, and you have obliged to the file-system requirements stated in this document, you are fine. However, structs as currently implemented, have a simple and minimalistic design and we suggest you continue using them as-is.

3 File System Manipulation

To implement the file system manipulation component of the API library, the following public APIs are needed (as described in `fat_files.h`):

1. `mini_file_dump(fs, file)` provides a human-readable presentation of the file's metadata, including its filename, size, blocks used to store its data, etc.

2. `mini_file_open(fs, filename, is_write)` will open a file for reading or writing. Once a file is opened, a `FAT_OPEN_FILE` structure is returned as a file handle that can be passed to other APIs that work with open files. This structure should at least include whether the file was opened in read or write mode, and what the current cursor position is inside the opened handle.

   Note that a file can be opened multiple times for reading, but can only be opened once for writing. The API should return `NULL` if it cannot open the file.

   Also note that opening a non-existing file for writing will automatically create it.

3. `mini_file_size(fs, filename)` will return the size of a file as an integer (in bytes). If the file does not exist, returns 0.

4. `mini_file_close(fs, open_file)` will close an open file handle.

5. `mini_file_delete(fs, filename)` will delete a file from the virtual disk. If the file is already open, it cannot be deleted. Returns true on success.

6. `mini_file_seek(fs, open_file, offset, from_start)` should change the position of the cursor in an opened file. Note that each open file handle has its own separate cursor.

   In relative mode (i.e., when `from_start` is false), the cursor should be moved by `offset` bytes, which can be positive or negative.

In absolute mode (when `from_start` is true) the cursor should be set to the provided offset.

The cursor cannot go beyond the beginning or end of the file. As such, the function should return true on success, and false on failure.

7. `mini_file_write(fs, open_file, size, buffer)` should write `size` bytes taken from `buffer` to the `open_file`.

   The write API should automatically add new blocks to the file as it is writing into it. It should also automatically update the file size as it writes data into it.

   The API returns the number of bytes successfully written. This should always match the `size` argument unless the virtual disk has no more blocks and is thus full.

8. `mini_file_read(fs, open_file, size, buffer)` should read `size` bytes from `open_file` and put it into `buffer`.

   The function should return the number of bytes successfully read, which only differs from `size` if the end of file is reached.

These APIs are defined in `fat_file.h`, and implemented in `fat_file.cpp` as stubs or skeletons. Similar to the disk manipulation APIs, there are several helper functions that are not part of the API, but strongly suggested to be implemented, as they help you break down the problem and have a more well-structured code.

Note that all `mini_file_*` APIs receive a `FAT_FILESYSTEM*`, i.e., a pointer to a FAT file-system created using the disk manipulation APIs, as the first parameter. This means that file manipulation APIs should be able to work on multiple different virtual file-systems in one program.

Again as with disk manipulation APIs, feel free to modify the structures or anything else, **as long as the public-facing APIs and their signature remains the same and can be called from `main.cpp` and work properly**.

4 Grading

- To help you get a sense of the correctness of your implementation, a test-suite is provided inside the code-base in file `main.cpp`. The test-suite runs multiple tests on the file-system, and if all of them are successful, gives you a score of 100. Otherwise it will inform you which tests failed and what score you have received. Please keep in mind that receiving a 100 from the provided test-suite **does not guarantee** that your will get full score as your final grade.

- A more extensive test-suite **different** from the one provided in `main.cpp` will be run against your APIs for final grading, and you will only receive points for the tests that pass. Please keep in mind that if you hard code value of variables into your implementation (e.g. block size), your implementation will not be able to pass the grading test suits.

- This grading scheme means that if your program crashes mid-run due to valid calls to the public API, the rest of the tests will not run and you will lose points for all of them. as such, pay extra attention so that your program does not crash. Checking the return

values from operating system function calls to make sure they have executed properly, is an important part of making sure your program does not crash. Even if your program does not work properly in some APIs, you would still pass some of the tests related to other APIs, as long as your program continues to run properly and does not crash.

- To run the accompanied test-suite, which will also provide you with the achieved score, run `make` to build the program (`g++` compiler is needed), and then run the produced binary.
- We will also look at the source code of your project to make sure the logic is implemented as intended. If the logic is faulty or results are hard-coded for the test cases, points will be deducted from the score attained in grading.

5 Deliverables

Submit the followings items packed in a zip file that is named after your usernames (i.e., `your-username1-your-username2.zip`) to blackboard:

- **(95 pts)** .h and .cpp files that implements the file system (`fat.h`, `fat.cpp`, `fat_file.h`, `fat_file.cpp`).
- Any supplementary files for your implementation (e.g. Makefile). It's ok to include `main.cpp` in the package.
- **(5 pts)** a `README.md` or `README.txt` file briefly describing your implementation, particularly which parts of your code work.
- Each team must create a github repository for the project and add the TA as a contributor (username is MandanaBM). Add a reference to this repository in your README.md file. We will be checking the commits during the course of the project and during the project evaluation.
- Do not include any binaries or any created virtual file-systems in your submission.
- We will randomly select a few of the teams to present and demo their project.

Good Luck!