Operating Systems – spring 2024
Tutorial-Assignment 11

Instructor: Hans P. Reiser

Submission Deadline: Monday, April 8, 2024 – 23:59

(Last assignment)

A new assignment will be published every week. It must be completed before its submission deadline (late policy for programming assignments: up to two days, 10% penalty/day)

Lab Exercisess are theory and programming exercises discussed in the lab class. They are not graded, but should help you solve the graded questions and prepare for the final exam. Make sure to read and think about possible solutions before the lab class.

T-Questions are theory homework assignments and need to be answered directly on Canvas (quiz).

P-Questions are programming assignments. Download the provided template from Canvas. Do not fiddle with the compiler flags. Upload your solution as a single zip file on canvas.

The name of the zip file has to be "assignment<XX>-<LOGIN1>-<LOGIN2>.zip", where <XX> is the assignment number and <LOGIN1>, <LOGIN2> are your skel login names. In each source code file, put your group number (if applicable) and names of all group members.

The topic of this assignment is the implementation of file systems.

The tutorial part is shorter this week, in order to have some time available for assisting with the last programming assignment.

Lab 11.1: File System Cache

a. What is the basic idea of a file system cache (buffer cache) in main memory?

b. Some operating systems implement read-ahead for the buffer cache (i.e., request blocks from disk that are subsequent to a block accessed by an application). Discuss pros and cons.

Lab 11.2: File System Recovery

Suppose your bitmap or list of free disk blocks has been corrupted due to a system crash. Is it possible to recover from this situation? Discuss in detail how (or why not) this is possible.

T-Question 11.1: File System Implementation

a. to be added **10 T-pt**

P-Question 11.1: File System Implementation

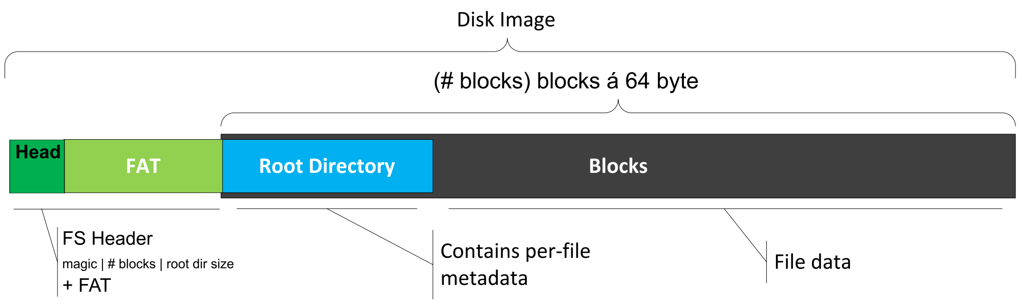
Download the template **p1** for this assignment from Canvas. You may only modify and upload the file `filesystem.c`.

In this assignment you will implement your own minimalistic FAT-like file system. The template comes with a disk image (`test.image`) that has been formatted with the assignment's file system.

Structure of the file system:

- Header (magic number, number of disk blocks, root directory size in bytes), see filesystem.h (all numbers are little-endian encoding)
- FAT (16-bit block numbers, as many as there are disk blocks)
- Data blocks, each 64 bytes (number of blocks defined in header)
- Two-level directory (the root directory, starting in disk block 0, contains files and subdirectories; subdirectories contain only files)

Graphical illustration of file system structure:


Disk Image
(# blocks) blocks á 64 byte
Head
FAT
Root Directory
Blocks
FS Header
magic | # blocks | root dir size
+ FAT
Contains per-file metadata
File data

Directory: Sequence of fixed-size entries with:

- Block number of first block of file (16 bit)
- File type (16 bit; 1=regular file, 2=directory, 255=deleted entry)
- Length of file (32 bit, in byte)
- Filename (8 byte)

Note that the root directory may not be stored in contiguous blocks (the first block is always block 0, but you have to use the FAT to find out what the next blocks are — if any)!

Open files are represented by file handles (pointer to the `OpenFileHandle` structure). This structure contains:

- `currentFileOffset`: Index of next byte to read within file (starts at 0)
- `currentBlock`: The file system block where the next byte is read
- `length`: Length of the file (copied from directory upon open)

Note: `currentFileOffset` is the full offset within the file. You can calculate the offset within the current block as `currentFileOffset % BLOCK_SIZE`.

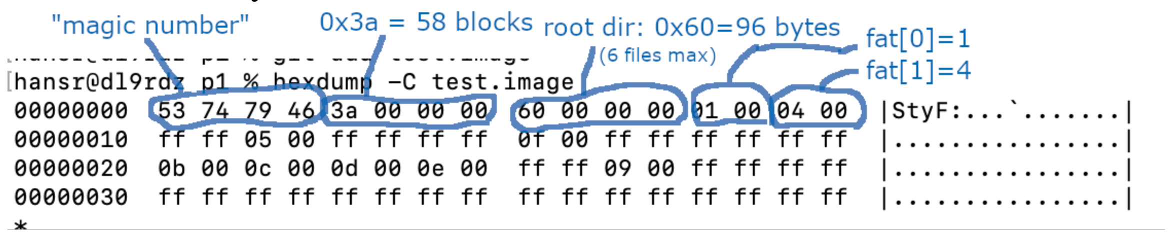
You'll find all relevant data structures and sizes in the template's header file. You can use the command `hexdump -C test.image` to view a hex dump of the disk image.

a. Implement a function that opens a full disk image. The function shall allocate a `FileSystem` data structure, read and store in memory the full header and FAT from the disk imagee, and return a reference to the data structure. Return `NULL` on any error. **2 P-pt**

```
        FileSystem *initFileSystem(char *diskFile);
```

(Note: This needs to be done in two steps. First you need to read the header (without FAT) to find out the size of the FAT, then you can allocate memory and read the full FAT.)

Illustration of file system structure:



```
hansr@dl9rdz p1 % hexdump -C test.image
00000000  53 74 79 46 3a 00 00 00  60 00 00 00 01 00 04 00  |StyF:...`.......|
00000010  ff ff 05 00 ff ff ff ff  0f 00 ff ff ff ff ff ff  |................|
00000020  0b 00 0c 00 0d 00 0e 00  ff ff 09 00 ff ff ff ff  |................|
00000030  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
```

b. Implement a function that returns 1 if more bytes can be read from an open file, 0 otherwise (i.e. end of file reached), based on the state of the file descriptor. **1 P-pt**

```
        int _hasMoreBytes(OpenFileHandle *handle);
```

c. Implement a function that reads the next byte from an open file, then updates the state of `OpenFileHandle`, then returns the byte that was read. If you reach the end of a block, make sure to update `currentBlock` using the file allocation table! **2 P-pt**

```
        char _readFileByte(OpenFileHandle *handle);
```

Note: The template already contains the implemenation of a function `readFile` that reads multiple bytes similar to the `read` system call, using your implementation of (b) and (c).

d. Implement a helper function that searches a directory (the directory itself is a file, represented by `dir`, whose content are directory entries) for a entry with name `name`. **2 P-pt**

- Use the `readFile` function to iterate over the `DirectoryEntry` structures of the directory represented by `dir`.
- Search for the requested file name (case sensitive).
- If the right entry is found, the `DirectoryEntry` for that file is copied to `dirEntry` and 0 is returned.
- Otherwise, the function shall return -1.

```
int _findDirectoryEntry(OpenFileHandle *dir, char *name,
                        DirectoryEntry *dirEntry);
```

e. Implement a function that opens a file from the file system by name. The template already contains a helper function `_openFileAtBlock` that creates a file handle structure Your function should perform the following operations: **3 P-pt**

- Create the file handle for the root directory (code already provided in the template).
- Ignore the `dir` parameter (for now).
- Use the `_findDirectoryEntry` helper function to search for `name`
- If the file name was not found, or if the file name was found, but entry is not of type FTYPE_REGULAR, return `NULL`.
- Otherwise, returns a new file handle for the requested file using `_openFileAtBlock` (using information from the directory entry).

```
OpenFileHandle *openFile(FileSystem *fs, char *dir, char *name);
```

f. BONUS QUESTION: Extend the function from the previous question for handling files in sub-directories. Functionality shall remain the same, with these modifications: **2 P-pt**

- if `dir` is not NULL, search for `dir` in the root directory and open that directory.
- Then, use this directory (instead of the root directory) to search for the file name

Total:
10 T-pt
12 P-pt