Operating Systems – spring 2022
Assignment 10

Instructor: Hans P. Reiser

Submission Deadline: Monday, March 28, 2022 – 23:59

A new assignment will be published every week, right after (or a bit before) the last one was due. It must be completed before its submission deadline.

T-Questions are theory homework assignments. **The answers to the assignments must be uploaded to Canvas (in the quiz).**

P-Questions are programming assignments. Download the provided template from Canvas. Do not fiddle with the compiler flags. Your solution shall compile and run on Linux systems (Intel x86_64). Solutions that fail with compiler errors can be rejected with 0 points. Additional tests are available on skel. Run them with
`/home/sty22/runStudTest.sh A10p1 <directory_with_your_code>`

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

T-Question 10.1: File System Implementation

a. Which table is used by the traditional MS DOS file system for chaining blocks in its linked list allocation? **1 T-pt**

true	false	
☐	☐	page table
☐	☐	file allocation table
☐	☐	index table
☐	☐	root directory

b. Assume you have a traditional Unix file system with inodes that contain 12 direct blocks, 1 single-indirect, 1 double-indirect and 1 triple-indirect block. The disk block size is 1024 bytes and the disk block address is stored as a 32 bit value. What is the maximum file size, and what is the largest file size that does not require any indirect blocks? (rounded to nearest MiB) **2 T-pt**

c. Assume you have a hard disk partition that stores 8 TiB of data blocks. The size of each block is 16 KiB (so in total you have 2^{29} blocks). The block address is stored in a 32 bit value. What is the size of a file allocation table for linked list allocation in this example? **1 T-pt**

d. A UNIX file system (with indexed allocation, inodes with 12 direct, 1 single-indirect, 1 double-indirect, 1 triple-indirect block, block size 4 KiB) stores 125000 files, each of them less than 1 KiB in size. How many inodes are used in total, and how many indirect index blocks with additional block addresses are used? **2 T-pt**

e. Assume you want to have a file system that supports *sparse files*, i.e., a file can have "holes" that do not occupy space on disk. For each of the three basic allocation methods (contiguous, chained, indexed), explain if (+how/why not) it is (easily) possible to implement sparse files! **3 T-pt**

f. How does the file system determine if an inode and thus the blocks allocated to the file can be deleted? **1 T-pt**

P-Question 10.1: File System Implementation

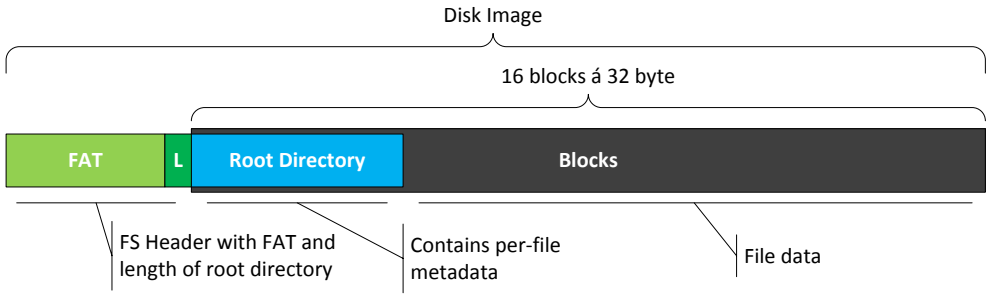
Download the template **p1** for this assignment from Canvas. You may only modify and upload the file `filesystem.c`.

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.

Properties of the file system:

- Fixed size: header, followed by 16 disk blocks with 32 bytes each.
- Single directory (the root directory), starting in disk block 0.
- Header: FAT (16 entries / 32-bit block numbers), size of root directory (bytes)

Graphical illustration of file system structure:



Directory: Sequence of fixed-size entries with:

- Block number of first block of file (32 bit)
- Length of file (32 bit, in byte)
- Filename (8 byte)

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 maps a full disk image into the address space of your program using the `mmap` system call. Cast the memory address of the disk image to a `FileSystem *` and return that address. Return `NULL` on any error. **3 P-pt**

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

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 function that opens a file from the file system by name. **4 P-pt**
The template already contains a helper function `_openFileAtBlock` that creates a file handle structure, and code to create the file handle for the root directory. Your function should perform the following operations:

- Use the `readFile` function to iterate over the `DirectoryEntry` structures of the root directory.
- Searches for the requested file name (case sensitive).
- Closes the root directory handle.
- If file name was not found, returns `NULL`.
- Otherwise, returns a new file handle for the requested file using `_openFileAtBlock` (using information from the directory entry).

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

Total:
10 T-pt
10 P-pt