Operating Systems – spring 2025
Tutorial-Assignment 11

Instructor: Hans P. Reiser

Submission Deadline: Wednesday, April 2, 2025 – 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 RU 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 Recovery

a. 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.

b. Assume you have a file system with indexed allocation and the following parameters:

- inode contains 10 direct blocks, 1 single-indirect block, 1 double-indirect block
- block size is 2048 bytes (2 KiB), and block numbers are stored as 32-bit number (4 byte)

(a) What is the maximum file size (in KiB) that can be stored using only direct blocks?

(b) What is the maximum file size (in KiB) that can be stored using the direct blocks and the single-indirect block?

(c) What is the maximum file size (in KiB) that can be stored on that file system?

c. Assume you have a file system with indexed allocation, storing allocation information as *extents*.

- Each extent is 48 bit physical block number and a 16 bit extent size (in blocks).
- Block size is 4096 byte.
- An inode contains information about four extents.

(a) What is the maximum file size (in KiB) that can be stored by a single file under those assumptions?

(b) Inspired by the ext4 file system: Assume that the inode can also specify for each of the four extent items in the inode whether it is actually a “direct extend” (as assumed so far), or an extent index (pointing to a disk block that contains additional extents (as many as fit into a *single* block). What is the maximum file size now?

T-Question 11.1: File System Implementation

a. Which data structure is used by a file system using indexed allocation for storing information about the data block numbers of a file? **1 T-pt**

true	false	
☐	☐	page table
☐	☐	file allocation table
☐	☐	index table
☐	☐	link counter

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 4 TiB of data blocks. The size of each block is 16 KiB (so in total you have 2^{28} 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 1 KiB) stores 125000 files, each of them less than 4 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. In a previous assignment (part T10.1a) you have (among other files) seen a file “out3.txt”, for which “ls” shows a size of around 4.6 TiB. Considering the nature of that file, can you efficiently store this file on a FAT file system? Justify your answer!

   Note 1: The file is still available at /home/sty254/A10/files/out3.txt on sty.

   Note 2: This question is only about storing the file efficiently. While FAT file systems can be inefficient for random access to a file, due to the fact that random seeks are time consuming, this is irrelevant for this question. **3 T-pt**

f. Why is the file name not stored in the inode? **1 T-pt**

P-Question 11.1: File System Implementation

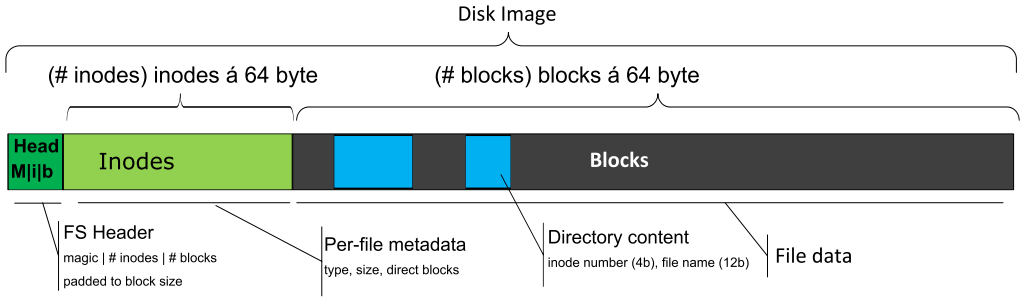
Download the template **p1** for this assignment from Canvas. You may only modify and upload the file `filesystem.c`.

In this assignment you will implement your own minimalistic inode-based 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 inodes, number of data blocks), see filesystem.h (all numbers are little-endian encoding)
- Inodes storing file type, file size, and block numbers of direct blocks (no indirect blocks are used). Number of inodes defined in header.
- Data blocks, each 64 bytes. Number of blocks defined in header.
- Two-level directories (the root directoy, stored in inode 0, contains files and subdirectories; subdirectories contain only files)

Graphical illustration of file system structure:



Directory: Sequence of fixed-size entries with:

- Inode number of file (32 bit)
- Filename (12 byte) — padded with '\0' if shorter

Note that the root directory may not be stored in contiguous blocks (the root directory's inode – this is always inode 0 – will store the list of blocks used by the directory. There are four (16 byte) directory entries per (64 byte) block.

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)
- `inodeNumber`: The inode number of the file
- `inode`: A copy of the full inode block (copy made when opening a file)

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 from the disk imagee, and return a reference to the data structure. Return `NULL` on any error. **2 P-pt**

```
FileSystem *initFileSystem(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. **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