File system implementation

MINI-PROJECT

To be submitted for evaluation to mooshak.di.fct.unl.pt until end of 2nd December 2025

This assignment is to be performed in **groups of two students maximum**. All solutions will be compared, and any detected frauds will cause all students to fail this course. Do not look at other students' code and do not show your own code to others. The **code must be identified with the students' names and numbers** and submitted via the Mooshak system using one of the students' individual accounts.

Objective

The goal of this assignment is to implement several operations in a simple file system inspired by UNIX-like file systems. This file system includes regular files and directories. You will work with an incomplete feature set of the provided file system. The following sections describe the file system's disk format, the provided code, and the tasks you must complete.

File system format

This project uses a simple file system (FS) stored in a simulated volume or disk that is implemented with a file. In this setup, reading or writing 'disk blocks' involves reading or writing fixed-size chunks of data to or from this file. Therefore, read and write operations always start at an offset that is a multiple of the 'disk block' size.

The FS format mapped onto the disk begins with a superblock, which describes the organization of the disk. Following that, there are one or more blocks that contain a bitmap representing used and free disk blocks, where each bit indicates whether the corresponding block is in use. After the bitmap, there are one or more blocks containing inode-like structures (as discussed in lectures). The rest of the disk is used for data blocks, which can store file contents and directory entries (dirents).

Figure 1 shows an example of a disk with a total of N blocks, initialized with the FS format. In this example, two blocks are allocated for the bitmap and two for the inodes table, though the exact layout may vary depending on the disk size.

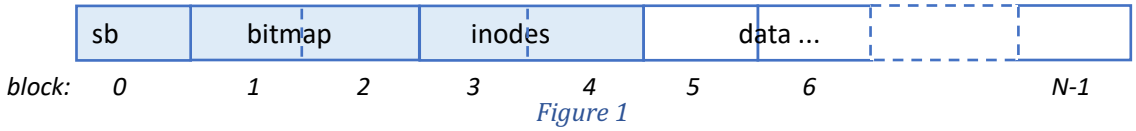

Figure 1

The file system layout details

The FS format details are described in the following text and are also represented by the C structures declared in the provided code. The disk is organized into 1K blocks in the following sequence:

- Block 0: the superblock, which describes the disk's organization using the following fields and sizes:
  - "magic number" (32bits): **0xF50F5025**, that is used to verify if the disk is a formatted FS;
  - total number of disk blocks (32bits);
  - number of blocks for the bitmap (16bits). The bitmap starts at block 1;
  - inodes start block (16bits), the block right after the last bitmap block;
  - number of blocks used for inode table (16bits);
  - first data block number. The block after the last inodes' block;
- Bitmap of used/free blocks: this map describes the disk usage, with each bit representing each disk block. If bit k is set to 1 it means disk block k is in use. For a disk of N blocks, the bitmap requires N bits. With blocks of 1024 bytes (1KB), each block can hold a bitmap up to 8K bits (8x1K). If the disk has

more than 8K blocks, the bitmap will span over multiple blocks as needed. Code in bitmap.c is provided for using this bitmap. There is no bitmap of used/free inodes (see inodes).

- Blocks with inodes: one or more blocks used to store the table of all the inodes. The number of inodes must be at least 25% of the number of disk blocks. Each inode is numbered by its position in this table. The inode 0 (zero) is reserved for the root directory and is always in use. An inode, when in use, describes a file or directory using the following fields:
    - Type (16bits) - if the inode is FREE, this field has the value 0; if it represents a directory, it has the value 4 (IFDIR) and, for a regular file, it has the value 8 (IFREG)
    - Number of links (16bits) – the number of links to this inode (names pointing to this inode)
    - Size (32bits) – the size of file or directory's content in bytes. The following indexes will point to the blocks needed to store that content size (i.e. if size is 1250 bytes and blocks are 1KB, two indexes will point to the two needed blocks)
    - An array of 11 direct indexes to data blocks (16bits each)
    - One indirect index (16bits) to a block containing more indexes to data blocks;
- Remaining disk blocks: used for storing the contents of files and directories.

For directories, their content will be an array of directory entries (*dirents*), each one using 64 bytes and containing:

- inode number (16bits)
- A 62 char array with the file or directory name, as a C string (ending with '\0');

Provided Code

The provided code includes a Makefile, so that it can be compiled by just executing the make command. This compiles and links all the source code:

- fso-sh.c – main program (see below). Uses functions from fs.c
- fs.c – file system implementation. This uses disk.c and bitmap.c.
- disk.c – device driver simulation. Offers functions for reading and writing blocks to the virtual disk.
- bitmap.c – bitmap of used/free blocks. Offers functions to set, clear and test bitmap bits.

The header files (.h) declare the public interfaces of each module. Also, two examples of disk images are included in files `small.dsk` and `medium.dsk`. Use then to test your program.

Shell to use the FS

A shell to browse and test the file system is implemented in fso-sh.c. This program can be invoked as in the example below that uses the `small.dsk` file as a disk:

```
./fso-sh small.dsk
```

One of the commands is *`help`*:

```
fso-sh> help
Commands:
    debug
    ls [<dirname>]
    create <filename>
    rm <filename>
    ln <filename> <newname>
    mkdir  <dirname>
    help or ?
    quit or exit
```

The pathnames (files or directories) can start with "/" or not, but are always relative to the root directory. May also include subdirectories separated by "/". Most commands are based on the functions offered by

the `fs.c` code (see description below and the comments in the source code). The debug command prints a dump of the FS structures in the disk that should help you in the debugging process.

Work to do

All fso-sh commands must work as intended but for that, the FS implementation in `fs.c` must be completed. Implement the features described in the present section.

The fs.c is the only file that you need to modify and is the file to submit to Mooshak.

This file includes the declarations of structures that define the types for superblock, inodes, and directory entries (*dirents*), which are used to represent these objects' data structures in memory variables. There are also several auxiliary functions, including the ones in bitmap.c and disk.c, that you may find useful. Take some time to become familiar with the provided C code before you begin.

The implementation in fs.c is incomplete, but we are not going to implement all the FS features, only the ones requested here. Also, these operations do not pretend to be system calls of an OS but are inspired by those so all `fs_` functions return -1 in case of error. Note that not all possible errors are described below.

Complete the following functions:

int fs_mkdir(char *dirname)

Creates a new directory. The directory should start empty, with a size of 0 bytes (there are no "." nor ".."). Returns the inode number allocated for this directory or -1 in case of error (possible errors may be that the *dirname* already exists, or some directory in the middle of the pathname doesn't exist, or there is no more free space, etc).

int fs_create(char *filename)

Creates a new regular file. Returns the inode number for this file or -1 in case of error (like when the *filename* exists, or some directory in the middle of the pathname doesn't exist, or there is no more free space, etc).

int fs_link(char *filename, char *newname)

Creates a new filename link from the existing file name. Returns the inode number of this file or -1 in case of error (like when the *filename* doesn't exist, or is a directory, or no more space, etc).

When needed, the previous functions should use the following policy for allocating a free dirent, inode, or data block: start from the lowest number and use the first available one.

int fs_unlink(char *filename)

Removes the filename. This means that its *dirent* entry should be marked as removed by replacing its inode number with FREE (note that inode zero is reserved for the root directory and cannot be used for any file, so can be used here to represent a deleted *dirent*). Also, inode's link count must be decremented and if it reaches 0 the file must be deleted, freeing its inode and all the indexed data blocks. Returns the inode number of this file or -1 in case of error (like when the *filename* doesn't exist, or is a directory, etc).

int fs_ls(char *dirname)

This function prints a list of the existing names in the given directory. If dirname is NULL or points to "" (empty string) the root directory is assumed. Suggestion: find the *inode* for *dirname* and then go through all indexed blocks, printing all the valid *direntries* (do not print the deleted ones). The following output is an example of running the *ls* command:

```
fso-sh> ls
listing dir / (inode 0):
ino:type:nlk    bytes name
  1:   F:  1        0 f0
  2:   D:  1      128 dir1
  3:   F:  1      617 file10
  4:   F:  2     5562 link.txt
 11:   D:  1    11520 d-big
. . .
```

Returns 0 or -1 on error (like when *dirname* is not a directory, etc).

Please note that the processing of indirect indexes is necessary in some operations. Examples:

- when growing a directory beyond some size when creating a new file or directory;
- when deleting a large file that uses indirect indexes.

If in your implementation only handles direct indexes and not the indirect ones (i.e., not when the file or directory occupies more than 11 blocks), you can still earn up to 85% of the total grade.

Suggestions:

After reviewing and understanding the provided code, begin by implementing the fs_ls function to list only the root directory. This directory is always stored in inode 0 (zero), so you can start by listing all the *dirents* in the data blocks indexed by inode 0. Some of the auxiliary functions provided will help you with that. Next, retrieve the inode information for each file or directory and print it as requested above.

The following step is to extend fs_ls so that it can accept a subdirectory name. To do this, you must first resolve the given name to its corresponding inode. After, to support full pathnames, you will need to implement traversal from the root directory, through intermediate subdirectories, until you locate the final name and its corresponding inode, which you will then list as above.

Once the `ls` command is working, you can use it together with the `debug` command to verify the changes made to the filesystem by your implementation of the other functions.

Bibliography

[1] "Operating Systems: Three Easy Pieces" Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau"

[2] Slides from FSO classes