FACULDADE DE
CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA
Fundamentos de Sistemas de Operação 24/25

File system implementation

MINI-PROJECT

To be submitted for evaluation to mooshak.di.fct.unl.pt until end of 4th December 2024

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 the discipline. 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. The simple file system includes files, directories, and inodes objects. 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 from or to this file. Therefore, read and write operations 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.

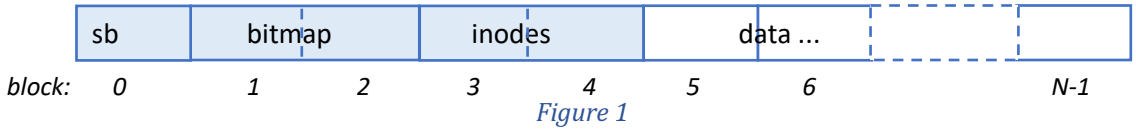

Figure 1

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 2K blocks in the following sequence:

- Block 0: the superblock, which describes the disk's organization using the following fields and sizes:
  - "magic number" (32bits): **0xF50F5024**, 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 in block 1;
  - inodes start block (16bits), the block right after the last bitmap block;
  - number of blocks for inode table (16bits). This is the number of blocks that accommodate a number of inodes of at least 10% the total number of disk blocks;
  - first data block number. The one right 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 2K blocks, each block can hold a bitmap up to 16K bits (8x2K). If the disk has more than 16K blocks, the bitmap will span over multiple blocks as needed. Code in bitmap.c is provided for using this bitmap.

- Blocks with inodes: one or more blocks used to store the table of all the inodes. 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 uses 32 bytes and, 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, has value 4 (IFDIR) and, for a regular file, has value 8 (IFREG)
    - Number of links (16bits) – the number of names to this inode (not used in this project)
    - Size (32bits) – file or directory contents size in bytes. The next indexes will point to the blocks needed so store this file or directory contents
    - An array of 11 direct indexes to blocks (16bits each)
    - One index (16bits) to a block containing more indexes;
- Remaining blocks: used for storing the contents of files and of directories.

For directories, their contents 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 bits.

The header files (.h) declare the public interfaces of each module. Also, some examples of disk images are included in files with `.dsk` extension.

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, which uses the `small.dsk` file as a disk:

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

One of the commands is *`help`*:

```
fso-sh> help
Commands are:
    debug
    ls [<dirname>]
    cat     <name>
    copyout <name> <filename>
    help or ?
    exit or quit
```

Most commands are based in the functions offered by the `fs.c` code. The command `copyout` will use `fs_open` and `fs_read` to copy a file from inside our virtual filesystem to a real file in our computer. The command `cat` is similar but will copy the file to the *standard output*.

Work to do

All previous commands must work as intended but for that, the FS implementation in `fs.c` must be completed. Implement and complete the features described in this section.

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

The implementation of the FS is incomplete, but we are not going to implement all the FS features, just 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.

The fs.c 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 private auxiliary functions, that you may find usefull. Please read the C code and all its comments.

Complete the following functions:

int fs_ls(char *dirname)

This function should print a list of the files in the given directory. Resolves the *dirname* to its *inode* number and then goes through all indexed blocks printing all the valid *direntries* like in the following exemplified output of the *ls* command:

```
listing dir / (inode 0):
ino:type bytes name
  1:D      128 dir1
  2:F      530 file1
  5:F     5562 file2
```

Returns 0 or -1 on error, like when *dirname* is not an existing directory.

int fs_open(char *pathname, int openmode)

Opens the given file for reading or writing, as given in the *openmode* parameter (the valid values for *openmode* are `O_RD` and `O_WR`, defined in `fs.h`). Resolves the *pathname* to its inode number. Creates an internal open file representation, that includes the offset initialized to 0, and returns a file descriptor that identifies that open file representation. Your implementation should support at least 512 simultaneous open files. This function can return -1 if the *pathname* does not represent a regular file, *openmode* does not contain a valid value or more than 512 files are already opened.

Suggestion: use an array of open files where each element is a structure with all the information about the open file: a copy of its inode, the openmode and current offset. The file descriptor will be the respective index of that array.

int fs_close(int fd)

Closes the given file descriptor, freeing its open file representation. Returns 0 if successful, or -1 in case of error, like when fd is not a valid open file.

int fs_read(int fd, char *data, int length)

This function reads at most length bytes, into data from the file represented by the given file descriptor, starting at its current *offset*. Returns the number of bytes effectively read that can be less than that length, if not enough bytes exit in the file after the current offset (it will return 0 at end of file). Can return -1 on error, like if fd is not a valid open file or if the file was not opened for reading.

Bibliography

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

[2] Slides from FSO classes