NOVA SCHOOL OF SCIENCE & TECHNOLOGY
DEPARTAMENTO DE INFORMÁTICA

# File system implementation

## MINI-PROJECT

**To be submitted for evaluation to mooshak.di.fct.unl.pt until 30th November 2023**

This assignment is to be performed in groups of two students maximum and any detected frauds among groups' assignments will cause failing the discipline. The code has to be submitted for evaluation via the Mooshak system using one of the students' individual accounts.

## Description

The goal of this assignment is to complete a few operations of a simple file system, which is inspired by some UNIX-like file systems (MinixFS and UFS). This is incomplete and do not use the full features of the described FS, like subdirectories or permission checking. The next sections describe the FS format and available code. These are followed by a section describing the tasks you must complete in this assignment.

## File system format

This work uses a very simple FS that is stored in a *volume,* like a disk, but simulated by a *file* where reading or writing *disk blocks* corresponds to reading or writing *fixed sized data chunks* from or to that file. Therefore, all read or write operations start at file's offset that are multiple of the disk block's size.

The FS format mapped on disk starts with a super-block describing the disk's organization. The next blocks contain a map of used/free disk blocks, where each bit represent if the corresponding block is in use. Finally, the remaining disk blocks are used for data, including the directory and files' contents. There are no subdirectories.

Figure 1 shows an overview of a particular disk with **N** blocks in total, after being initialized with the format command. The figure shows the disk's organization according to the superblock's fields as described next.

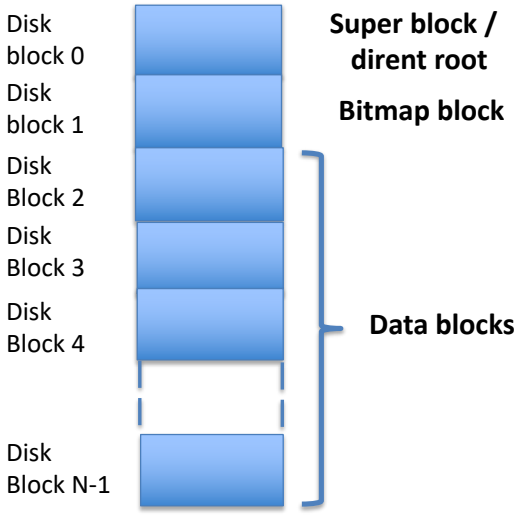


*Figure 1*

## The file system layout

The FS organization is described in the following points:

- The disk is organized in 2K blocks.

- Block 0 is the super-block and describes the disk organization. The majority of values in the superblock are unsigned integers (4 bytes):
    - "magic number" (**0xfafafefe**) that is used to verify if the file image is a FS formatted disk
    - total number of blocks in the all file system (field named *nblocks*)
    - size of the bit map in bytes, round up (field named *bmapsize_bytes*)
    - number of blocks the bitmap occupies on disk; usually one block (field named *bmapsize_blocks*)
    - number of the first data block; usually block number two (field named *first_datablock*)
    - total number of data blocks (field named *data_size*)
    - block size in bytes used in this filesystem (field named *blocksize*)
    - one directory entry describing the root directory (field named *rootdir*)
- Block 1 has the disk blocks bit map. This map of used blocks defines the disk's occupation and it uses one bit per disk block. If a disk has N blocks, the bit map will occupy N bits. If bit k has value 1 it means disk block k is in use. Bits representing the blocks occupied by superblock and the bitmap itself are always 1.
- Blocks starting at the first data block are used for storing the contents of the root directory and of the regular files. The rootdir dirent structure in the superblock, has in its blk array the data blocks for the root directory. Each of these blocks contains the directory entries (dirent) for all files and includes inode-like information. Each dirent (type *struct fs_dirent*) has the following format:
    - a flag identifying if the entry is valid
    - the name of the file represented in this entry, represented as a C string
    - the file size in bytes
    - an array with the direct references to data blocks occupied by the file. This array has *POINTERS_PER_E*NT elements and each element have a disk block number.

## Disk emulation

The disk is emulated in a file and it is only possible to read or write data chunks of 2K bytes, each starting at an offset multiple of 2048. File disk.h defines the API for using the virtual disk. The implementation of the virtual disk API is in the file disk.c.

## File system operations

The file fs.h describes the public operations to manipulate the file system, used by fso-sh.c. The implementation is in fs.c. These are incomplete and do not use the full features of the described FS. Also, these operations do not pretend to be the system calls operations of an OS. Notice that the number of the dirent is used like a file descriptor in read and write operations and that the file's offset must be given upon some operation's calls.

`void fs_debug()` – prints the FS layout information

`int fs_format()` – formats the disk (includes creating the superblock)

`int fs_mount()` – mounts the filesystem (reads the superblock)

`int fs_lsdir()` – lists the contents of the root directory

`int fs_create(char *name)` – creates a new file

`int fs_open( char *name)` – resolves the file name to its dirent number

`int fs_read( int inumber, char *data, int length, int offset )` – reads length bytes, starting at offset, from the file described by the provided dirent number

`int fs_write( int inumber, const char *data, int length, int offset )` – writes length bytes, at given offset, to the file described by the provided dirent number

## A Shell to operate the FS

A shell to manipulate and test the file system is available to be invoked as in the example below:

```
$ ./fso-sh image20
```

where the argument is the name of the file/disk supporting the file. One of the commands is *help*:

```
fso-sh> help
Commands are:
    format
    mount
    umount
    debug
    ls
    create  <name>
    cat     <name>
    copyin  <name_of_file_in_the_local_file_system> <name_of_fs_file>
    copyout <name_of_fs_file> <name_of_file_in_the_local_file_system>
    help
     exit
```

The commands `format`, `mount`, `umount`, `debug`, `ls`, and `create` correspond to the functions previously described or to functions you can inspect in the fs.c code. Do not forget that a file system must be formatted before being mounted, and a file system must be mounted before creating, reading or writing files. The commands that use the functions `fs_read()` and `fs_write()` are also available:

- `copyin` copies a file from the local file system to the simulated file system.
- `copyout` executes the opposite
- *`cat`* reads the contents of the specified file and writes it to the standard output (uses *copyout*)

Example:

```
>> copyin /usr/share/dict/words  testfile
```

## Work to do

Implement and complete the following features:

***int fs_format()***

Complete the format operation by filling all the superblock structure's fields before writing it to disk.

***int fs_lsdir()***

This function should list the files in the directory, i.e it prints all the valid entries like in the exemplified output of the *ls* command:

```
fso-sh> ls
entry size  name   blocks
0     1000  f1.txt 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1     2000  f2.txt 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2     6000  f3.txt 5 6 7 0 0 0 0 0 0 0 0 0 0 0 0
3     0     f4     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

***int name2number(char *name)***

This function resolves the file name to its directory entry number. This function is needed by *fs_create()* and *fs_open()*.

***int fs_read( int inumber, char *data, int length, int offset )***
***int fs_write( int inumber, char *data, int length, int offset )***

These functions read/write from/to the file described by the dir entry with number *inumber*, starting at the given *offset*, a number of bytes specified in the *length* parameter.

The implementation of the operations above is in file *fs.c*. ***It is the only file that you need to modify and the file to submit to Mooshak.***

## Recommendations

**Do not forget to handle error situations**: your code must deal with situations like a file too big or too many files, and so on. Please handle these situations gracefully and do not terminate your program abruptly (i.e. identify possible error situations and return -1 in that case).

**The metadata must be synchronously updated to the disk**: every time you modify superblock or the directory, you must write such block to disk.

**How to prepare a new empty disk:**

- Invoking the fso-fs with a non-existing file and number of blocks:

  ```
  $ ./fso-sh filename size
  ```

  If the file does not exist, it will be created with the indicated size. After that, you can format and mount that file system. Once created, any file system can be reused again by just using it:

```
$ ./fso-sh filename
```

## Available code

Download the proj.zip archive file from CLIP containing the files `Makefile`, `fso-sh.c`, `fs.h`, `fs.c`, `bitmap.h`, `bitmap.c`, `disk.h`, and `disk.c`.

The program's modules are organized according to Figure 2:

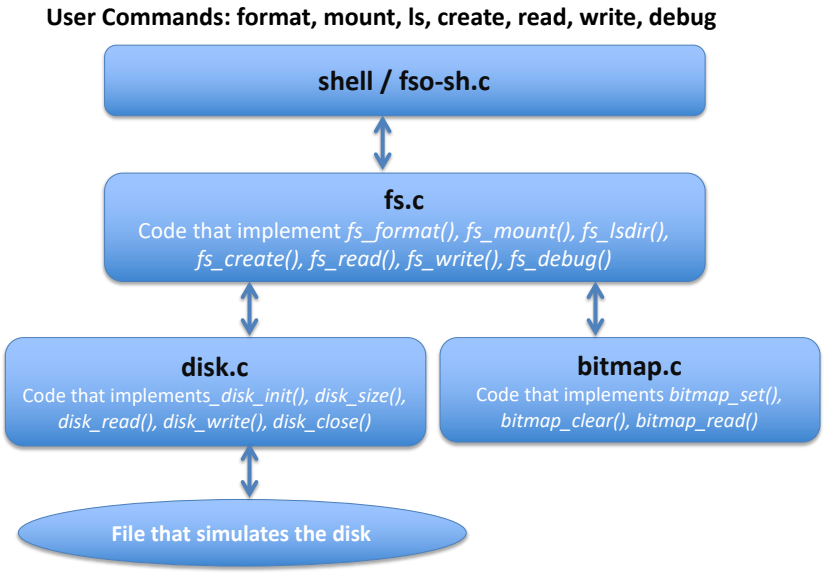


*Figure 2*

## Bibliography

[1] Sections about persistence (chapters 36 and 39) of the recommended book, "Operating Systems: Three Easy Pieces" Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau"

[2] http://pages.cs.wisc.edu/~remzi/OSTEP/file-intro.pdf

[3] Slides from FSO classes