Fundamentos de Sistemas de Operação

MIEI 2017/2018

Homework Assignment 4

Deadline and Delivery

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 (http://mooshak.di.fct.unl.pt/~mooshak/) using one of the students' individual accounts -- the deadline is 23h59, **December 11th, 2017** (Monday).

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). These are 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 provided code. These are followed by a section describing the tasks you must complete in this assignment.

The file system

The file system (FS) is stored in a *disk* simulated by a *file* from which is possible to read or write *disk blocks* corresponding to *fix sized data chunks*.

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 inodes and a map of used/free disk blocks, where their bytes represent if is in use or free. The following blocks contain the inodes table, where the used inodes contain meta-data of sub-directories as well as existing files in the FS and references to their data blocks. Finally, the remaining disk blocks are used for data (directories' or files' contents).

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.

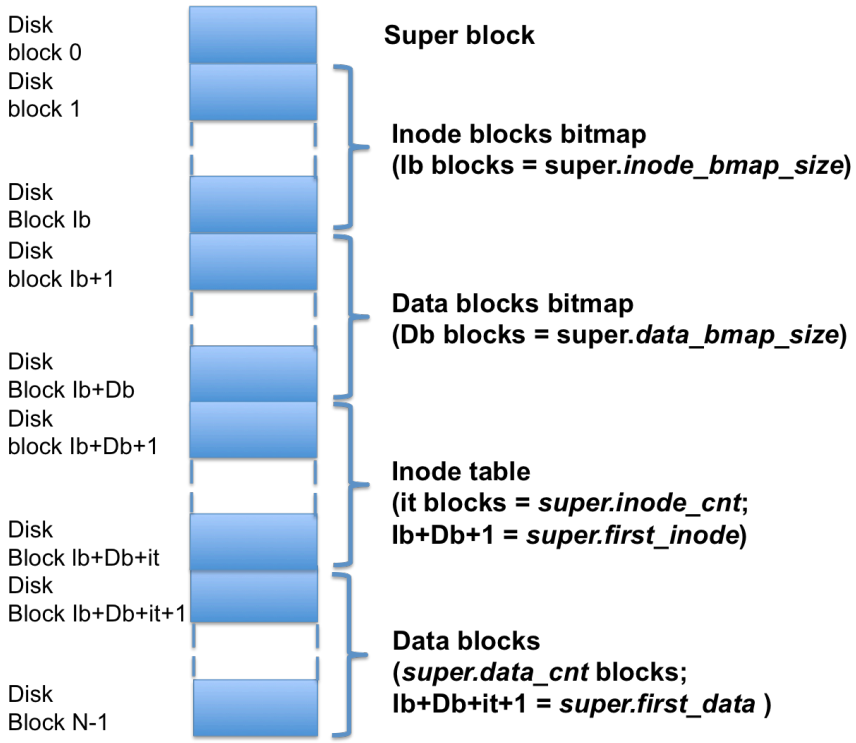

Figure 1

The file system layout

The *FS organization* is described in the following points:

- The disk is organized in 4K blocks.

- Block 0 is the *super-block* and describes the disk organization. All values in the superblock are *unsigned short integers* (2 bytes) and they are placed in the super-block *in the following order*:
  - "magic number" (**`0xf0f0`**) that is used to verify if the file image is a FS formatted disk
  - status of the last unmount operation (if it was correctly performed or not)
  - size of the byte map for inode blocks (field named *inode_bmap_size* -> Ib in Figure1)
  - size of the byte map for data blocks blocks (field named *data_bmap_size* -> Db in Figure1)
  - number of the first block of the inodes (field named *first_inode*)
  - total number of inode blocks table (field named *super_inode_cnt* –> it in Figure 1)
  - number of the first data block (filed named *first_data*)
  - total number of data blocks (field named *data_cnt*)
- Block 1 and possibly the following ones have the inodes byte map. If byte *n* has value 1 it means inode *n* is in use.
- Blocks starting at 1+*size of the byte map for inode blocks* have the disk blocks byte map. The map of used blocks defines the disk's occupation and it uses one byte per block. If a disk has *N blocks*, the byte map will occupy *N bytes*. If byte *n* has value 1 it means disk block *n* is in use. Byte 0 and the following ones, representing the metadata blocks, are always 1.
- Blocks starting at the first inodes' block contain inodes. Each **inode in the inode table** contains a set of values, most with two bytes, in the following order:
  - node type defines if the inode represents a directory, a regular file, etc
  - access mode permissions
  - owner uid
  - group id
  - file size (in bytes – 4 bytes)
  - last modification time (4 bytes)
  - number of name links
  - a vector with the direct references to data blocks occupied by the regular file, directory, etc; the vector has 20 elements and each element occupies two bytes; these two bytes have value of the disk block number
  - one number for the disk block number containing the first indirect data block references
  - another number for the disk block containing the second indirect data block references

- Blocks starting at the first data block are used for storing the contents of the directories, regular files, etc. Directories have the directory entries (dirent) with the following format:
  - Two bytes for the inode number containing the meta-information for this entry
  - The name of the file/directory located in this entry, represented as a C string

Disk emulation

The disk is emulated in a file and it is only possible to read or write data chunks of 4K bytes, each starting at an offset multiple of 4096. File *disk.h* defines the API for using the virtual disk:

```
#define DISK_BLOCK_SIZE 4096
int  disk_init( const char *filename, int nblocks );
int  disk_size();
void disk_read( int blocknum, char *data );
void disk_write( int blocknum, const char *data );
void disk_close();
```

The implementation of the virtual disk API is in the file *disk.c.* The following table summarizes the operation of the virtual disk:

Function	*Description*
`int disk_init( const char *filename, int nblocks );`	This function must be invoked before calling other API functions. It is only possible to have one active disk at some point in time.
`int disk_size();`	Returns an integer with the total number of the blocks in the disk.
`void disk_read( int blocknum, char *data );`	Reads the contents of the block disk numbered *blocknum* (4096 bytes) to a memory buffer that starts at address *data*.

`void disk_write( int blocknum, const char *data );`	Writes, in the block *blocknum* of the disk, a total of 4096 bytes starting at memory address *data*.
`void disk_close();`	Function to be called at the end of the program.

File system operations

The file *fs.h* describes the public operations to manipulate the file system. 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 inode 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 detailed information including, when mounted, a listing of all the files

`int fs_format()` – formats the disk (includes creating the root dir (/)

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

`int fs_mkdir()` – create a directory (not implemented)

`int fs_create(char *name)` – creates a new file (for now, only names in the root dir)

`int fs_open( char *name)` – resolves the file name to its inode number (for now, only names in the root dir)

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

`int fs_write( int inode, const char *data, int length, int offset )` – writes length bytes, at given offset, to the file described by the provided inode 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 image.20 20
```

where the first argument is the name of the file/disk supporting the file system and the second is its number of blocks in case you are creating a new file system. One of the commands is *`help`*:

```
>> help
Commands are:
    format
    mount
    debug
    fsck
    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`*, *`debug`*, *`fsck`*, and *`create`* correspond to the functions with the same suffix previously described or that you can inspect in the 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.

Some 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`* execute 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

Download the src-tpc4.zip archive file from CLIP.

Implement the following features:

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 file described by *inumber* inode, starting at the given *offset*, a number of bytes specified in the *length* parameter.

The inode supports two references to two equal indirect blocks. Complete the implementations to use de indirect blocks. This feature will allow files with more than 80Kbytes (20x4K). You can save files until (20+2K)*4K = 8272K using the first indirect block, and (20+2K+2K)*4K = 16464K using also the second indirect block. Similarly, when adding a new link (dirent) to a directory (fs_create, fs_mkdir), use indirect blocks when needed.

int fs_open(char *name)

int fs_create(char *name)

These functions resolve the file name to its inode. In case of fs_create, a new inode is allocated and initialized to describe the new file. Complete the implementations to support also subdirectories. The name will be an absolute pathname like in the Unix filesystems, using "/" as separator. The *direntries* and *inodes* must be followed and checked if the *inode* represents a directory, until reaching the regular file. Continue to ignore the owner, group and permissions mode as is. You should implement the **fs_mkdir** so that new directories can be created. There is no need to create '.' and '..' in each new directory.

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

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 the error code -1 in that case).

The metadata must be synchronously updated to the disk: every time you modify a metadata structure in memory you must write such structure to disk.

How to prepare a new empty disk:

- Invoking the fso-fs with a non-existing file

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

  If the file does not exist, it will be created with the indicated size in blocks. 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 src-tpc4.zip archive file from CLIP containing the files `Makefile`, `fso-sh.c`, `fs.h`, `fs.c`, `disk.h`, and `disk.c`.

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

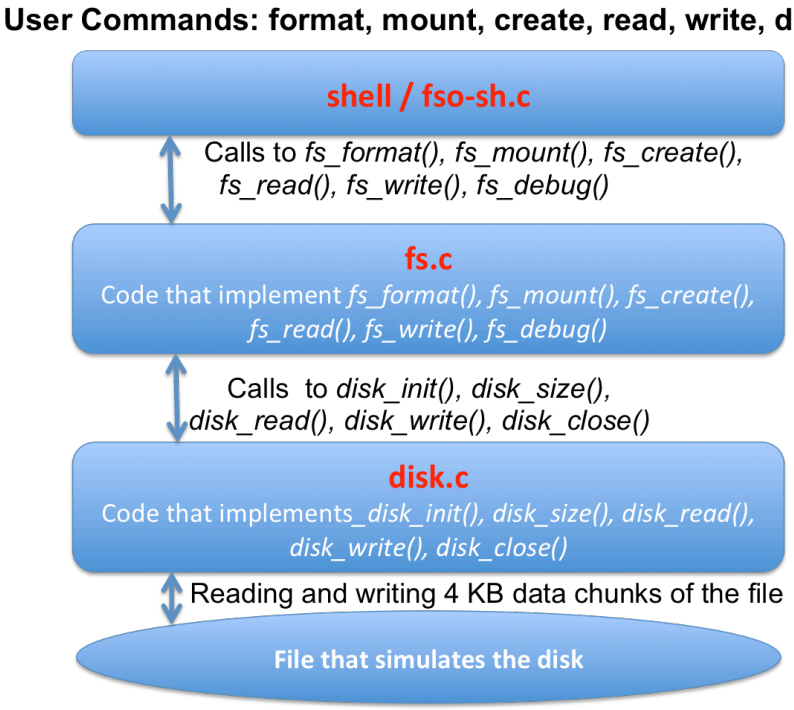

Figure 2

Bibliography

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