Fundamentos de Sistemas de Operação

MIEI 2014/2015

Programming Assignment 2

Deadlines

The programming assignment 2 (PA2) will be developed in the classes of the weeks starting 25th November, 2nd December and 9th December. The deadline for submitting PA2 (report and code) is **Monday, December 15th – 23:59**. Instructions for submission will be communicated in due time.

Very important note about the number of students per group

The programming assignment must be done by groups of 2 students. **Only with a strong justification and explicit authorization of the instructor, groups with one student will be accepted. This authorization must be granted not after December, 2nd.**

Assignment objectives

- To consolidate the knowledge about how a file system works.
- To add competences in the use of the C programming language.

This assignment can be solved in any environment where a C compiler is available.

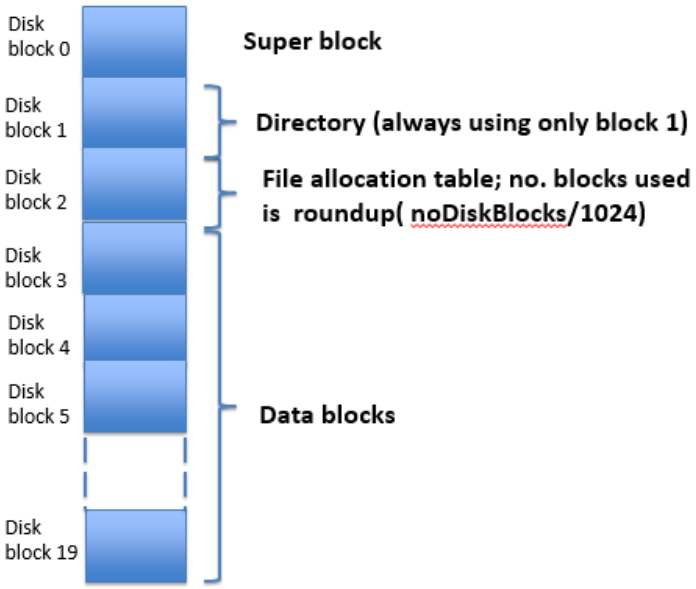
MIEI-02 file system[1]

This programming assignment concerns the development of a file system similar to Windows FAT; of course, many simplifications are made. Students will receive a set of files with code; one of the files is very incomplete and we ask you to fill the blanks in it. Students should not change the contents of the other files.

As expected the file system is stored in a disk; a Linux (or Windows or Mac OS X) file simulates the disk. Reading and writing blocks corresponds to reading and writing fixed size chunks of data from / to the file; all reads and writes start at an offset multiple of the block size.

MIEI-02 format

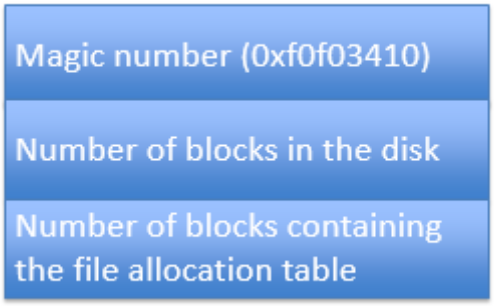
The following pictures show a disk with 20 blocks after being initialized with the MIEI-02 format:



[1] This assignment is based on a project proposed in April, 2005 by Prof. Douglas Thain of Notre Dame University, Illinois, USA

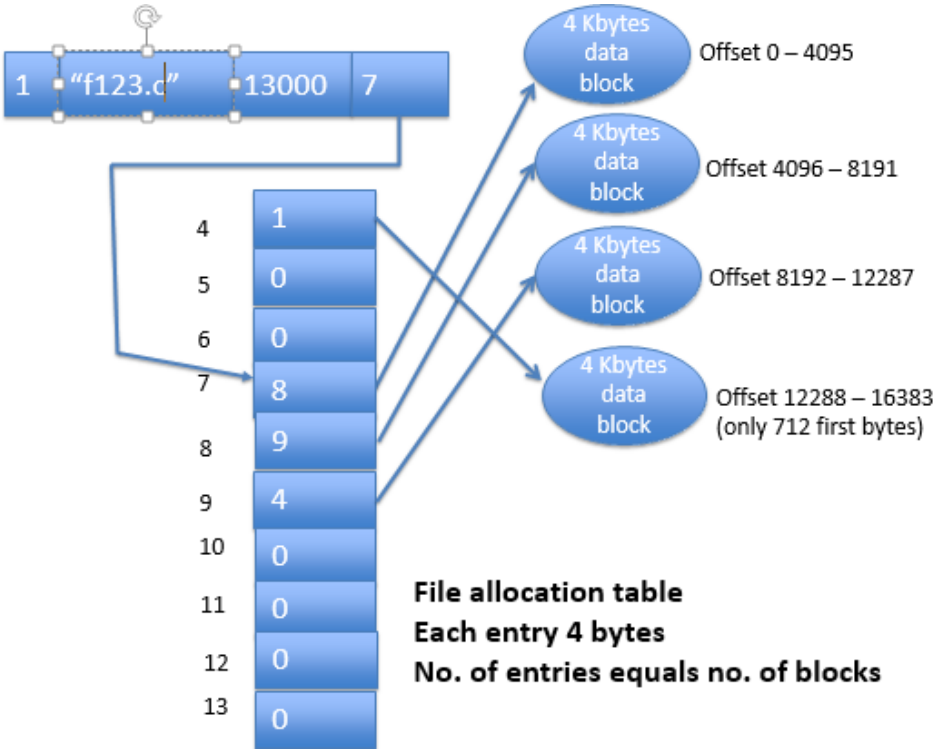
The *MIEI-02 file system* organization can be summarized in the following points

- **The disk is organized in 4K blocks. The file system layout is the following**
  - Block 0 has the super-block that describes the disk organization. In this block only the first 12 bytes are relevant.
    - First an integer (four bytes) must be initialized with a "magic number" (0xf0f03410) that is used to verify if the disk contains a valid file system
    - The second unsigned integer indicates the size of the disk in blocks (*nblocks*). Namely, 20 in the example.
    - The third unsigned integer has the number of blocks (*nfatblocks*) used by the file allocation table. In the case above, 1



  - Blocks used for storing the contents of the files (*nblocks - 2 – nfatblocks*)

- **Each entry in the directory has 16 bytes with the following contents**:
  - A boolean (unsigned char) indicating if the entry is free (0) or occupied (1)
  - A string with the file name (6 bytes maximum, terminated with 0)
  - An unsigned integer for storing the file size in bytes
  - An unsigned integer for storing the number of the first block of the file



- **The disk does not contain an explicit bit map of free/occupied blocks**. The file allocation table (FAT) performs that role: positons whose content is 0 correspond to free blocks; otherwise the block is occupied. When a disk is mounted the file allocation table is copied to main memory.
- **The FAT also contains information about the blocks that have data from a given file *f***. The directory entry with the name *f* contains the address of the first block A1 of the file; the entry A1 of the FAT contains the address A2 of the second block and so on; the end of the file is marked by a FAT entry which contents is 1 (EOFF).
- **Files are named by a string with 1 to 6 characters terminated by a byte set to 0.**

Disk emulation

The disk is emulated by a file; one can only read or write 4K bytes that start in a offset that is a 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 following table summarizes the virtual disk operation:

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

The implementation of the virtual disk API is in the file *disk.c*.

File system operations

File `fs.h` describes the operations that manipulate the file system:

```
int  fs_format();
void fs_debug();
int  fs_mount();
int  fs_create( char *name);
int  fs_delete( char *name );
int  fs_getsize( char *name );
int  fs_read( char *name, char *data, int length, int offset );
int  fs_write( char *name, const char *data, int length, int offset );
```

In the following table the actions corresponding to each file system operation are described:

int fs_format()

Creates a new file system in the current disk, wiping out all the data. Block 0 will contain the magic block, Block 1 is used for directory; all directory entries are marked as free. The file allocation table starts at block 2 and occupies (nblocks / 1024) disk blocks rounded to the next integer. If one tries to format a mounted disk, *fs_format* should do nothing and return an error. Returns 0 in case of success and -1 if an error occurs.

void fs_debug()

Displays information about the current active disk. The expected output is something like:

```
superblock:
     magic number is valid
     1010 blocks on disk
     2 blocks for file allocation table
File "xpto1":
     size: 4500 bytes
     Blocks: 103 194
File "hello.c":
     size 11929 bytes
     Blocks: 105 109 210
```

This should work either the disk is mounted or not. If the disk does not contain a valid file system – indicated by the absence of the magic number in the beginning of the super block – the command should return immediately indicating why.

int fs_mount()

Verifies if there is a valid file system in the current disk. If there is one, reads the directory and the file allocation table to RAM. Note that all the following operations will fail if there is not a mounted disk. Returns 0 on success and -1 on fail.

int fs_create(char *name)

Creates an entry on the directory describing an empty file (length 0) with name *name*; the directory is updated in RAM and in the disk. Returns 0 on success and -1 on fail.

int fs_delete(char *name)

Removes the file *name*. Frees all the blocks associated with name and updates the file allocation table in RAM and in the disk; after this releases the directory entry. Returns 0 in case of success; -1 in case of error

int fs_getsize(char *name)

Returns the length in bytes of the file associated with the specified name. In case of error returns -1.

int fs_read(char *name, char *data, int length, int offset)

Reads data from a valid file. Copies *length* bytes of the file specified by *name* to the address specified by the pointer *data*, starting at *offset* bytes from the beginning of file. Returns the total number of read bytes. This returned number can be less than the value specified in *length* if the end of file is reached. In case of error returns -1.

int fs_write(char *name, const char *data, int length, int offset)

Writes data to a file. Copies *length* bytes from the memory address *data* to the file identified by *name* starting in file offset *offset*. In general, this will imply the allocation of free blocks. The function returns the effective number of bytes written, which can be smaller than the value specified in *length*, for example if the disk becomes full. Returns -1 in case of error.

The implementation of these operations is in file *fs.c*. This file is incomplete.

Shell to manipulate the *miei-02* file system

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

```
% ./fs-miei02 image.20 20
```

where the first argument is the file supporting the file system and the second is the number of blocks. One of the commands is *help*:

```
fs-miei02> help
Commands are:
    format
    mount
    debug
    create
    delete  <name>
    cat     <name>
    getsize <name>
    copyin  <name_of_file_in_the_local_file_system> <name_of_miei_fs2_file>
```

```
copyout <name_of_miei_fs2_file> <name_of_file_in_the_local_file_system >
dump    <block_number_with_text_contents>
help
exit
```

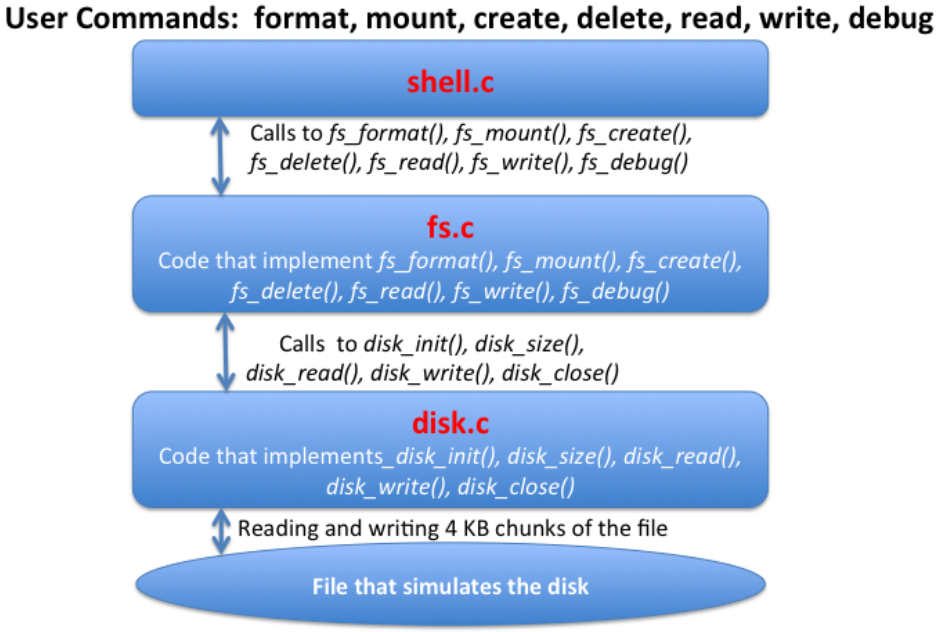
Commands *format, mount, debug, create and delete* correspond to the functions with the same suffix described above. Please do not forget that a file system must be formatted before being mounted; and a file system must be mounted before creating, deleting, reading and writing files.

Some commands that use the functions *fs_read()* and *fs_write()* are also available:

- *cat* reads the contents of the specified file and writes it in the standard output
- *copyin* copies a file from the local file system to the *miei-02* file system
- *copyout* makes the opposite

Example:
```
fs-miei-02> copyin /usr/share/dict/words xpto
```

The program in development is organized according to the following figure:



Code supplied

Download the *fs-miei02.zip* file from CLIP (***Documentação de Apoio-> Outros***) that contain *Makefile, shell.c, fs.h, fs.c. disk.h, disk.c.*

Notes about the supplied code

Data structure corresponding to the super-block:

```
#define DISK_BLOCK_SIZE    4096
#define FS_MAGIC           0xf0f03410


typedef struct fs {
        int magic;
        int nblocks;
        int nfatblocks;
        char filler[DISK_BLOCK_SIZE - 3*sizeof(int)];
} super_block;

super_block mb;          // global variable filled when the disk is mounted
```

Data structure corresponding to the single directory:

```
#define VALID 1
#define NON_VALID 0
#define MAX_NAME_LEN 6
#define N_DIR_ENTRIES (DISK_BLOCK_SIZE / sizeof(dir_entry))

typedef struct{
        unsigned char used;
        char name[MAX_NAME_LEN+1];
        unsigned int length;
        unsigned int first_block;
} dir_entry;

dir_entry dir[N_DIR_ENTRIES];      // global variable, filled when the disk is mounted
```

Data structure corresponding to the FAT:

```
#define N_ADDRESSES_PER_BLOCK (DISK_BLOCK_SIZE / sizeof(int))
#define FREE 0
#define BUSY 2
#define EOFF 1

unsigned int *fat; // space allocated after mounting the disk and knowing
                   // the number of blocks
```

Work to do

The only file that you need to modify is *fs.c* in order to add the functionality described. Do not change the other files.

Some advices

Follow the order below in the implementation.

1. Start by the implementation of fs*_debug*, *fs_ format* and *fs_mount* functions.
2. Next do the creation and removal of files; start by using only empty files. Test file system images will be available.
3. Implement *fs_read*
4. At last implement *fs_write*.

Do not forget to handle error situations: Your code must deal with situations like disk full, full directory, full FAT, and so on. Please handle these situations gracefully and do not terminate abruptly your program.

How to prepare a new empty disk: There are two simple ways of doing this:

- Using the *miei02-fs* shell: invoking the *miei02-fs* with a non-existing file

  ```
  ./miei02-fs    filename size
  ```

  If the file does not exist, it will be created with the indicated size in blocks.

- Using the following shell command (in the example a disk with 20 blocks is created; *man dd* for details)

  ```
  dd if=/dev/zero of=newImage count=20 bs=4k
  ```