Fundamentos de Sistemas de Operação

MIEI 2013/2014

Programming Assignment 2

Deadlines

Programming assignment 2 (PA2) will be developed in the classes of the weeks starting 18th November, 25th November and 2nd December. The deadline for submitting PA2 (report and code) is ***Monday, December 9th – 9:00****. Please see the instructions for submission at the end of this document.*

Assignment objectives

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

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

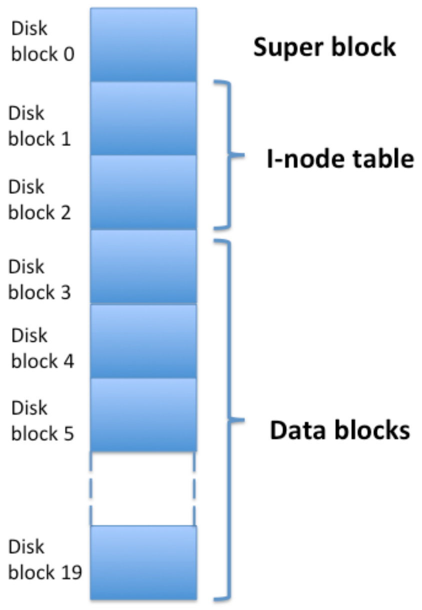
MIEI-01 file system[1]

In this programming assignment a file system similar to the ones used in UNIX/Linux will be developed. Many simplifications are assumed and students will receive a lot of code; the task will be to add code that implements some missing functionalities.

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 correspond to reading and writing fixed size chunks of data from / to the file all starting at an offset multiple of the block size.

MIEI-01 format

The following pictures depicts how a disk with 20 blocks appears after being initialized with the MIEI-01 format:

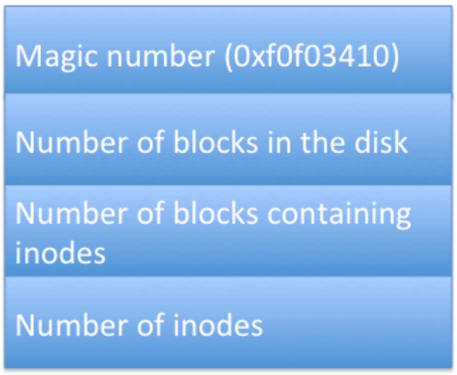


MIEI-01 file system organization can be summarized in the following points

- **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 16 bytes are relevant

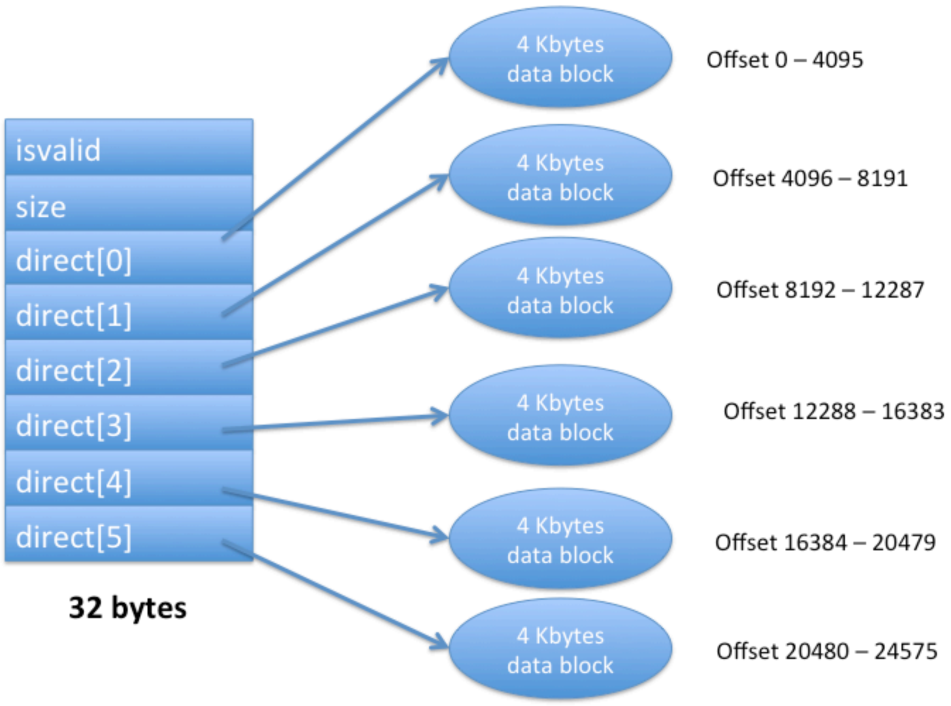
[1] This assignment is based on a project proposed in April, 2005 by Prof. Douglas Thain of Notre Dame University, Illinois, USA

        - 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*). 20 in the example.
        - The third unsigned integer has the number of blocks (*ninodeblocks*) used by the i-node table. In the case above, 2
        - The fourth unsigned integer has the total number of entries of the i-node table. In the example 2 * (4096 / 32) = 256


Magic number (0xf0f03410)
Number of blocks in the disk
Number of blocks containing inodes
Number of inodes

    - *Ninodeblocks* that contain the i-node table
    - Blocks used for storing the contents of the files (*nblocks -1 – ninodeblocks*)

- **Each i-node entry has 32 bytes with the following contents**:
    - Boolean (unsigned integer) indicating if the i-node is free (0) or occupied (1)
    - Unsigned integer for storing file size in bytes
    - Array of 6 addresses of blocks where the file contents is stored. This means that our file system cannot store files bigger than 6*4K, i.e., 24 Kbytes



- **The disk does not contain an explicit bit map of blocks free/occupied**. When a disk is mounted this block occupation bitmap is built in RAM by going through the i-node table and setting to 1 all the bits that correspond to blocks referenced in a entry
- **Files are named by an unsigned integer that corresponds to the index in the i-node table**.

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 virtual disk operation:

Function	Description
`int  disk_init( const char *filename, int nblocks );`	Function that must be invoked before calling other functions of the API. One can only have one active disk at the same 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 call in 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();
int  fs_delete( int inumber );
int  fs_getsize( int inumber );
int  fs_read( int inumber, char *data, int length, int offset );
int  fs_write( int inumber, 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. 10% of the disk blocks are allocated to the i-node table; all the entries of the i-node table are marked as free. If one tries to format a mounted disk *fs_format* should do nothing and return an error.

`void fs_debug()`

Display the i-node table contents of the current active disk. The expected output is something like:

```
superblock:
     magic number is valid
     1010 blocks on disk
     101 blocks for inodes
     12928 inodes total
inode 3:
     size: 4500 bytes
     Blocks: 103 194
inode 5:
     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 superblock to RAM and builds the map of free/occupied blocks using the contents of the i-node table. Note that all the following operations will fail if there is not a mounted disk.

int fs_create()

Creates a new i-node corresponding to an empty file (length 0). *The block pointers should be initialized to zero*. In case of success return the i-node number allocated; -1 in case of error.

int fs_delete(int inumber)

Removes the file *i-node*. Frees all the blocks associated with the i-node and updates the map of free/occupied blocks; after this releases the i-node itself. Returns 0 in case of success; -1 in case of error

int fs_getsize(int inumber)

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

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

Reads data form a valid i-node. Copies *length* bytes of the file specified by *inumber* to the address specified by the pointer *data*, starting *offset* bytes from file beginning. Returns the total number of read bytes. The number returned can be less than the asked if file ends. In case of error returns -1.

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

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

The implementation of these operations is in file *fs.c*. Some operations are missing in this file.

Shell to manipulate the *miei-01* file system

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

```
% ./fs-miei01 image.5 5
```

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-miei01> help
Commands are:
    format
    mount
    debug
    create
    delete  <inode>
    cat     <inode>
    copyin  <file> <inode>
    copyout <inode> <file>
    help
    exit
```

Commands *format, mount, debug, create and delete* correspond to the functions with the same suffix described above. Please don´t 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.

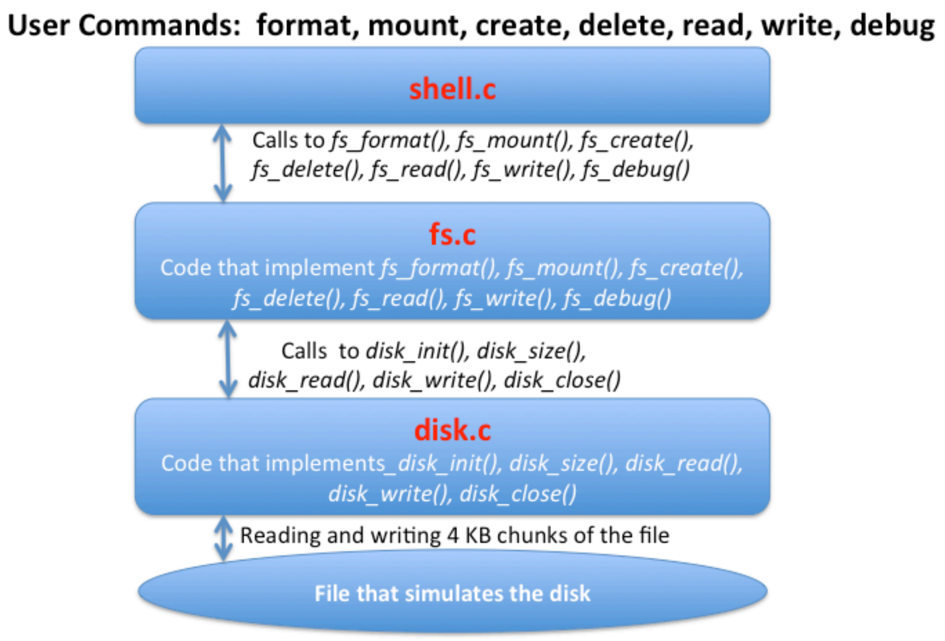
Some commands that use the functions *fs_read()* and *fs_write()* are also available:

- *cat inumber* reads the contents of the specified file and writes it in the standard output
- *copyin* copies a file form the Linux file system to the *miei-01* file system
- *copyout* makes the opposite

Example:

```
miei-01> copyin /usr/share/dict/words 10
```

The program that we will develop is organized according to the following figure:


User Commands: format, mount, create, delete, read, write, debug
shell.c
Calls to fs_format(), fs_mount(), fs_create(), fs_delete(), fs_read(), fs_write(), fs_debug()
fs.c
Code that implement fs_format(), fs_mount(), fs_create(), fs_delete(), fs_read(), fs_write(), fs_debug()
Calls to disk_init(), disk_size(), disk_read(), disk_write(), disk_close()
disk.c
Code that implements_disk_init(), disk_size(), disk_read(), disk_write(), disk_close()
Reading and writing 4 KB chunks of the file
File that simulates the disk

Code supplied

Download the following files from CLIP (***Documentação de Apoio-> Outros***): *Makefile, shell.c, fs.h, fs.c. disk.h, disk.c*.

For testing purposes you can also download files that contain valid file system initialized according to the *miei-01* format: *image.5* (20K), *image.20* (80K) and *image.200* (800K).

Notes about the code supplied

Data structures corresponding to superblock and i-node table:

```
#define DISK_BLOCK_SIZE     4096
#define FS_MAGIC            0xf0f03410
#define INODES_PER_BLOCK    128
#define POINTERS_PER_INODE 6

struct fs_superblock {
        int magic;
       int nblocks;
       int ninodeblocks;
       int ninodes;
};

struct fs_inode {
       int isvalid;
       int size;
       int direct[POINTERS_PER_INODE];
};
```

A disk block of 4096 bytes contains 128 inodes:

```
  struct fs_inode inodes[128];
```

Each disk block with data will corresponds to:

```
   char data[DISK_BLOCK_SIZE];
```

A disk block with 4K can have three different contents: a superblock, 128 inodes, or a data block. How to represent the data returned by *disk_read()*. A possible solution is to use the C language construct *union*. A union looks like a *struct* but corresponds to different data organizations, all occupying the same memory space:

```
union fs_block {
      struct fs_superblock super;
      struct fs_inode      inodes[128];
      char                 data[4096];
};
```

In this case the size of *fs_block* union is 4096 (the biggest of the variants). Declaring a union *fs_block* like

```
    union fs_block block;
```

allows the use of the union like in the following examples

```
  disk_read(0,block.data);
 x = block.super.magic;    // using the block 0 of disk as the super block
 disk_read(21,block.data);
 x = block.inodes[4].size;  // using block 21 as a part of the i-node table
```

Work to do

The only file that you need to modify is *fs.c* in order to add the functionality described. Don´t change the other files.

Some advices

Follow the order below in the implementation.

1. Start by doing fs*_debug, fs_ format* and *fs_mount* functions. Try these commands with the example images
2. Next do the creation and removal of i-nodes; use only empty files. Again test with the file system images available.
3. Implement *fs_read*
4. At last implement *fs_write*.

Identify common functions For example you will need functions for reading and writing i-nodes from / to disk. Prepare two functions to do this:

```
      void inode_load( int inumber, struct fs_inode *inode ) { ... }
      void inode_save( int inumber, struct fs_inode *inode ) { ... }
```

Don´t forget to handle error situations: Your code must deal with situations like disk full, full i-node table, too big files, etc.. 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 *miei01-fs* shell: invoking the *miei01-fs* with a non-existing file

  ```
  ./miei01-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
  ```

Submission instructions

The work should be made by groups of two students. Each group definition is responsibility of its members. Individual work can only be accepted if is **first** authorized by the student’s teacher. The work deliverables are:

- The source code for *fs.c*.
- A small report (in PDF format with no more than 4 pages) describing implementation options, assumptions made, and possible limitations of your system.

These should be archived in a ZIP-file named “AAAAA-BBBBB.zip” were “AAAAA” and “BBBBB” are the student’s numbers (AAAAA < BBBBB). Send this file by email to your lab instructor with the subject:

```
FSO – Trabalho 2: alunos XXXXX nº AAAAA e YYYYY nº BBBBB
```

(put your group colleague in Cc). The instructor will acknowledge the reception; if you don’t receive a confirmation message after 48h contact your instructor.