Fundamentos de Sistemas de Operação

MIEI 2017/2018

Laboratory session 9

Objectives

To consolidate the knowledge on how a *file system* (*FS*) works via the implementation of the listing of a directory's contents of a very simple file system.

The file system[1]

This class uses a very simple FS that is stored in a *disk*, as expected. This disk is simulated as a *file* where reading or writing *disk blocks* corresponds to reading or writing *fixed sized data chunks* from or to the file. As a consequence, all read or write operations start at a file's offset that is multiple of the disk block's size.

The *FS format* contains a *super-block* describing the disk's organisation, a directory that is also the inodes table containing meta-data of the existing files in the FS and occupied blocks, and a set of available data blocks (either in use or free).

Figure 1 shows an overview of a particular disk with *20 blocks* and two blocks for the directory/inodes table, after being initialized with the *format* command.

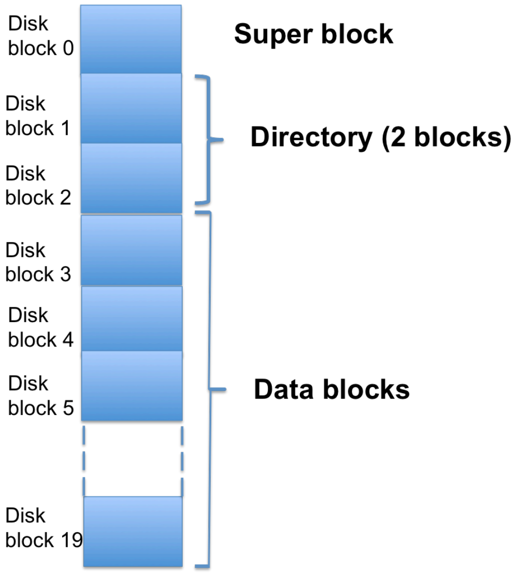

Figure 1

The file system layout

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

- **The disk is organized in 4K blocks.**

A block may either contain the superblock (block 0), the directory/inodes table (block 1 and the next ones accordingly to information in superblock), or data:

- <u>Block 0</u> is the *super-block* and describes the disk organization:
    - The first value is an unsigned integer (four bytes) that must be initialized with a "magic number" (**`0x00f00baa`**). It is used to verify if the disk contains a valid *FS*
    - The second value is also an unsigned integer defining the size of the disk in blocks (*nblocks*)

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

        - The third value (unsigned integer) defines the number of blocks for the directory/inodes table.
        - The fourth value (unsigned integer) defines the maximum number of inodes/directory entries.
    - Block 1 and possibly the following ones have the directory/inodes table.

    Each director entry is also like a inode and uses 128 bytes containing the following information:

        - A validation byte defining if the inode contains valid information (value one) or not (value zero)
        - Name, a sequence of bytes (63 bytes maximum) identifying a file's name
        - Size, (unsigned int) defining the size of the file
        - File blocks, (unsigned int) pointers to the file's blocks (15 max)
    - Blocks numbers used for storing the contents of the files

- **The used/free block bit map**

The map of used blocks is not in the disk -- it is build in memory when mounting the FS and defines the disk 's occupation. It uses one char per block (value zero if a block is free or one if it is occupied) meaning it is not a real bitmap. If a disk has *N blocks*, the byte map will occupy *N bytes*.

File system operations

The file *fs.h* describes the operations that manipulate the file system (notice that the inode is used like a file descriptor):

```
void fs_debug();

int  fs_format();

int  fs_mount();

int  fs_create( char *name);

int  fs_open( char *name);

int  fs_delete( char *name );

int  fs_read( int inode, char *data, int length, int offset );

int  fs_write( int inode, const char *data, int length, int offset );
```

Work to do – listing the contents of the directory

Download the src-L09.zip archive file from CLIP.

Implement the *listing of all files* in the `fs_debug` function, as well as the `fs_open` and the `fs_delete` operation.

The following text describes the actions corresponding to each FS operation:

`void fs_debug()`

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

```
superblock:
    20 blocks
    2 dir/inode blocks
    64 inodes/dirents
**************************************
inode   size    name blocks
0    12176     foo  3 4 5 0 0 0 0 0 0 0 0 0 0 0 0

**************************************
```

This should work either the disk is mounted or not. If the disk does not contain a valid file system, i.e. the magic number is not present in the beginning of the super block, the command should print *"Non-valid filesystem"* and return immediately.

`int  fs_open( char *name )`

Searches an entry on the directory describing the named file. Returns the inode/dirent number on success and -1 on fail.

int fs_delete(char *name)

Removes the file *name* and frees all the blocks associated with it and updates the byte map in RAM. Subsequently releases the directory entry in disk.

This function returns 0 in case of success and -1 in case of error.

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

Disk emulation

The disk is emulated in a file and it is only possible to read or write data chunks of 4K bytes each that start 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.

A Shell to operate the FS

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

```
$ ./fso-L09 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
    open    <name>
    create  <name>
    delete  <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*, *create* and *delete* 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, 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 to the standard output
- *copyin* copies a file from the local file system to the simulated file system
- *copyout* execute the opposite

Example:

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

Available code

Download the src-L09.zip archive file from CLIP containing the files `Makefile`, `fso-L09.c`, `fs.h`, `fs.c`, `disk.h`, and `disk.c`.

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

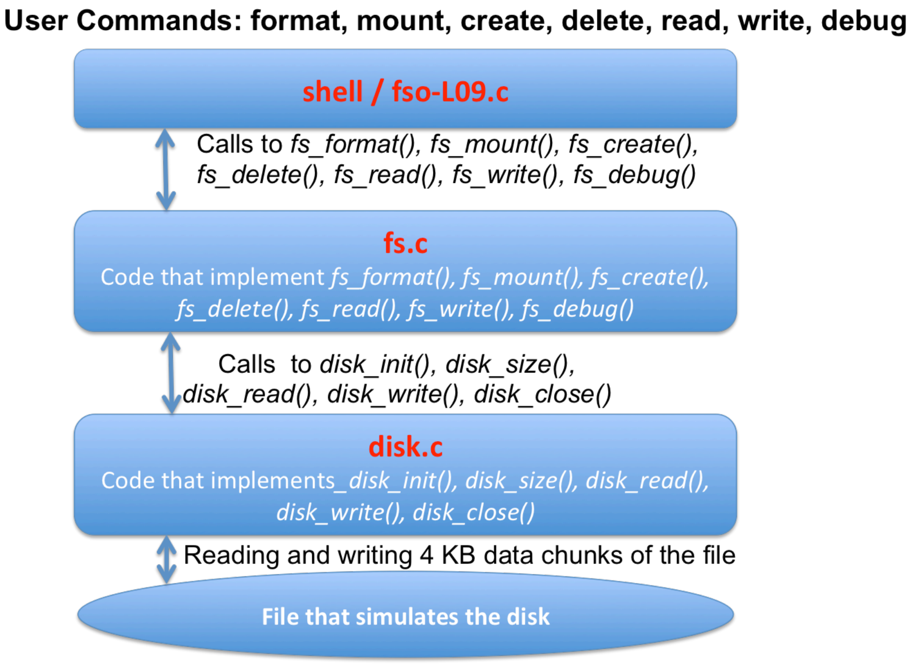
User Commands: format, mount, create, delete, read, write, debug
shell / fso-L09.c
Calls to fs_format(), fs_mount(), fs_create(), fs_delete(), fs_read(), fs_write(), fs_debug()
fs.c
Code that implement fs_format(), fs_mount(), fs_create(), fs_delete(), fs_read(), fs_write(), fs_debug()
Calls to disk_init(), 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 data chunks of the file
File that simulates the disk

Figure 2

Notes about the code

The data structures corresponding to the FS are:

```
#define DISK_BLOCK_SIZE    4096
#define FS_MAGIC           0x00f00baa

#define POINTERS_PER_INODE 15  // 64/4 - 1 = 15 --> max file size = 15*4K
#define INODES_PER_BLOCK 32    // 4K/128=32
#define MAXFILENAME    63      // 128 bytes per dirent-> 1+63+64

#define VALID 1          // constants to manage the inodes entries
#define NON_VALID 0
```

```
#define FREE 0x00      // constants to manage the byte map
#define NOT_FREE 1

struct fs_superblock {
        uint32_t magic;         // contains the magic number when formatted
        uint32_t nblocks;       // FS size
        uint32_t ninodeblocks; // number of blocks for inodes+directory
        uint32_t ninodes;       // max number of inodes/dir entries
};

struct fs_superblock rootSB; // superblock of the mounted disk

struct fs_inode {       // an inode is also a directory entry
        uint8_t isvalid;
        char name[MAXFILENAME];
        uint32_t size;
        uint32_t blk[POINTERS_PER_INODE];

};

// a block may contain the superblock, inode entries, or data.
union fs_block {
        struct fs_superblock super;
        struct fs_inode inode[INODES_PER_BLOCK];
        char data[DISK_BLOCK_SIZE];
};

unsigned char * blockBitMap; // Map of used blocks (1char=1block, not a real bitMap)
```

Bibliography

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