Fundamentos de Sistemas de Operação

MIEI 2018/2019

Mini Project

Deadline and Delivery

This assignment is to be performed in groups of two students maximum and any detected frauds 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 the students' accounts. The deadline is 22h00, **November 27th, 2018** (Tuesday).

Introduction

The goal of this assignment is to implement a few operations of a simple file system, inspired by the old CP/M, a very simple (and rudimentary) operating system for 8- and 16-bits microcomputers from the 70's. CP/M (or *Control Program for Microcomputers*) could run just one program at a time and was the inspiration to the MS-DOS and similar systems. It could use floppy-disks of until 360Kbytes and, later, hard-disks with at most 32MBytes. Giving these capacities, saving space when storing data on such devices was key and there was also no need for elaborated data structures and metadata used by todays filesystems.

This work consists in two phases, where only the first is mandatory, corresponding to 80% of full grade. The second phase can give the final 20%.

The next sections describe the FS format to implement, that is not the same of CP/M. These is followed by a description of the provided code and tasks you must complete in this assignment.

The old file system

Our old file system (OFS), or Our File System, will be stored on a *disk device* simulated by a *file* from which is possible to read or write *disk blocks* corresponding to *fix sized data chunks*.

The *OFS format* mapped on disk starts with a *super-block* describing the filesystem organization on disk. The next blocks contain the files directory (just one, there are no subdirectories) and data files. Each directory entry (or ***dirent***) contains meta-data describing each file and their data blocks. Note that this filesystem doesn't have *inodes* as some of that information is included in each *dirent*. If a file is so big that one *dirent* isn't enough to keep all of its data blocks, more than one *dirent* can be used (see below) and those *dirents* are also called file *extents*. For the first phase assume that there is no extents and all the files are small.

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 starting with the superblock, that is described next.

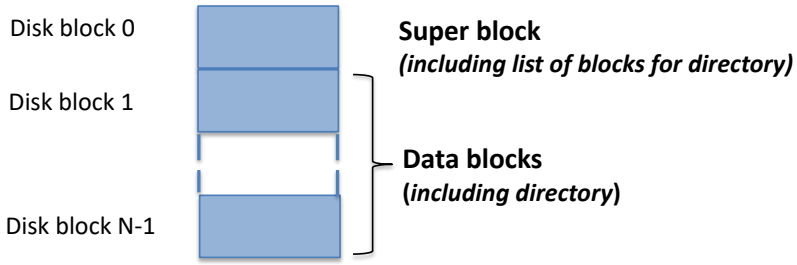

Figure 1

The file system layout

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

- The disk is organized in 1K blocks.
- Block 0 — is the *super-block* and describes the disk's organization. All numerical values are *unsigned short integers* (2 bytes, little-endian). The information placed in the super-block is the following:

- "magic number" (**`0x0f0f`**) that is used to verify if the file image is a OFS formatted disk
- total number of blocks in this file system (max supported device size: 64 Mbytes)
- 12 bytes with disk label, a name given by the user when the device was formatted, with ASCII upper case leters or numbers, with spaces to the right (this is not a C string! There is no '\0').
- List of 503 directory block numbers (2 bytes each; unused positions are represented with zero)

- Block 1 and the following ones — contains data, starting with the first block of the directory.
- Each block of the directory contains 32 directory entries (*dirent*) that uses 32 bytes each, with the following information:

```
ST F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 E1 E2 S1 S2
B1 B2 B1 B2 B1 B2 B1 B2 B1 B2 B1  B2  B1 B2 B1 B2
```

  - ST (1 byte): entry status/type, defining if it represents a regular file (0x10), a removed entry or not in use (0x0), or a file *extent* (0xff).
  - F… (11 bytes): the file name in the form of ASCII uppercase letters, numbers, '.' or '_'. Unused chars will be filled with spaces at right (this is not a C string! There is no '\0').
  - E1,E2 (2 bytes): *extent* counter. To phase 2, if a file takes up more blocks than can be listed in one directory entry, it is given multiple entries. If this is the first *dirent* (0x10 in the ST byte) represents how many extra *extents* are used by this file; if it's an *extent* entry (0xff in the ST byte), represents its sequence number (starting with 1 for the first *extent* after file's *dirent*). Each *extent* must include the file name.
  - S1,S2 (2 bytes): number of bytes in the **last** *extent* (it is this *dirent* if file is less than 8K). The file size is given by *num-extents*8K+this size*.
  - B…. : 8 unsigned short ints (2 bytes) with the block numbers used for content (each *dirent* allows a file of at most 8Kbytes). Using *extents* (more *entries*), a file can grow up to fill the entire disk.

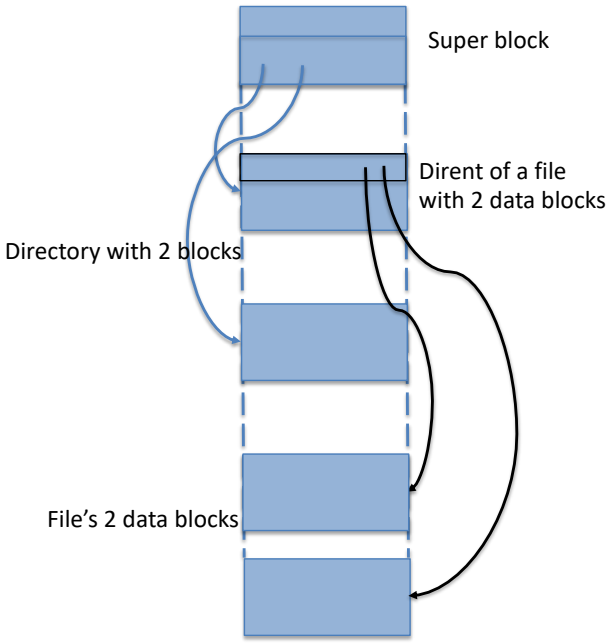

Figure 2

Figure 2 gives an example where two blocks are used for directory entries and a *dirent* represents a file using two data blocks.

Disk emulation

The disk is emulated in a file and it is only possible to read or write data blocks of 1K bytes. File *disk.h* defines the API for using the virtual disk. 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. If file doesn't exist, is created with *nblocks* size. If it exists, is used as is and *nblocks* is ignored.

`int disk_size();`	Returns an integer with the total number of blocks in the disk.
`void disk_read( int blocknum, char *data );`	Reads the contents of the disk block numbered *blocknum* (1024 bytes size) to a memory buffer that starts at address *data*.
`void disk_write( int blocknum, const char *data );`	Writes, to the block *blocknum* of the disk, 1024 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. Those operations do not pretend to be the system calls of a real OS. Notice that the file name will be used as an identifier for read and write operations (there is no open or create) and that the file's offset must be given upon those operation's calls.

`void fs_debug()` – prints debug information including a listing of all the files in directory (fs_dir bellow)

`int fs_dir( )` – lists all files in the directory and their sizes

`int fs_format(char *label)` – formats the disk (initializes the relevant disk blocks for OFS)

`int fs_mount()` – mounts the filesystem (checks disk device, reads the superblock and initializes the used/free blocks map)

`int fs_read(char *name, char *data, int length, int offset )` – reads length bytes, starting at offset, from the file named

`int fs_write(char *name, const char *data, int length, int offset )` – writes length bytes, at given offset, to the file named. If the file doesn't exist, it must be created. If it exists, data is written over the old content. The file must grow when needed (new data blocks and extents must be allocated)

`int fs_delete(char *name)` – deletes the file named (frees entries from directory and all its data blocks)

A Shell to operate the FS

A shell to manipulate and test the file system is available and can 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.

This shell implements several commands. One of the commands is *`help`*:

```
>> help
Commands are:
    format
    mount
    debug
    dir
    cat     <name>
    copyin  <name_of_file_in_the_host_file_system> <name_of_fs_file>
    copyout <name_of_fs_file> <name_of_file_in_the_host_file_system >
    delete <name>
    help
    exit
```

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 *`format`*, *`mount`*, *`dir`*, and *`debug`* correspond to the functions with the same suffix previously described. Some commands that use the functions *`fs_read()`* and *`fs_write()`* are also available:

- *`copyin`* copies a file from the host file system to the simulated file system
- *`copyout`* copies a file from the simulated file system to the host

- `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 – Phase 1

Download the src-miniproj.zip archive file from CLIP. Complete the following functions implementation:

void fs_dir()

void fs_mount()

int fs_delete(char *name)

Implement also the following functions:

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

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

These functions read/write from/to the named file, starting at the given *offset*, the number of bytes specified in the *length* parameter. Assume, for now (phase 1), that all the files are less than 8K and use just one *dirent.* In this case S1S2 field gives you the file size (there is no *extent*).

When writing, it's possible that new data blocks must be allocated and added to its *dirent*. All changes must update the used/free map in memory and the respective disk blocks used for directory and file contents.

All the operations above are in file *fs.c*. ***It is the only file that you need to modify!***

In this assignment *length* can be bigger than one disk block, but it's always less than 8K. Also when writing, the offset is always inside current file size or equal to filesize (so there is no need to create gaps or empty zones in your file).

An example image (**`A.img`**) is provided for your first tests.

Work to do – Phase 2

The implementation must take into account file *extents*. These must be searched and followed for reading or writing the relevant data blocks. Also, remember that when adding new data blocks to file, new *extents* may also need to be added to the file, and its *dirent* information updated.

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. Examples: new blocks used by directory, new *dirents*/*extents* in directory, files content, etc.

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
  ```

Bibliography

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