Fundamentos de Sistemas de Operação

MIEI 2019/2020

Mini Project

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 must 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 4 th, 2019** (Wednesday).

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 operations are incomplete and do not use the full features of the described FS, like subdirectories or permission checking. The next sections are organized in two parts. The first part includes the sections defining the FS format and the provided code and a section describing the tasks you must complete in this first part of the assignment. The second part is related with implementing a *block disk cache* and the subsequent sections describe the modifications you have to perform to your code.

The file system FS1920

The file system FS1920 is stored in a *disk* simulated by a *file* where is possible to read or write *disk blocks* corresponding to *fixed size data chunks*.

A disk formatted with the *FS1920 format* has the following organization:

- A *super-block* describing the disk's organization
- The next ***I*** blocks contain the i-node table, where the used i-nodes contain meta-data on existing files in the FS. This table occupies 10 % of the disk data blocks (rounded up)
- Finally, the remaining disk blocks are used for the files' contents

Figure 1 shows an overview of a disk with **20** *blocks* in total, after being initialized with the *format* command.

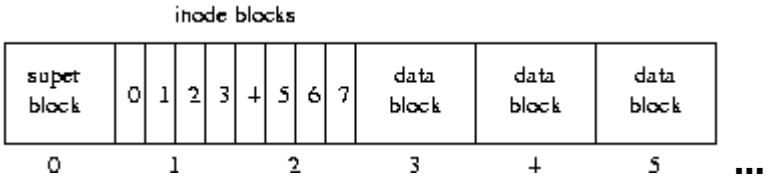

Figure 1

The file system layout

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

- The disk is organized in 4K blocks
- Block 0 is the *super-block* and describes the disk organization. All values in the superblock are *unsigned integers* (4 bytes) and they are placed in the super-block *in the following order*:
  - *fs_magic* "magic number" (`0xf0f03410`) - used to verify if the file image is a FS formatted disk
  - *nblocks* total number of blocks (equals the disk size in blocks)
  - *ninodeblocks* number of blocks reserved to store i-nodes; this must be nblocks/10 rounded to the nearest upper integer
  - *ninodes* number of i-nodes
- Blocks starting at 1 and finishing in *ninodeblocks* contain inodes. Each **inode in the inode table** has the following contents, where each element is an *unsigned integer* (4 bytes) and is placed in the inode *in the following order*:
  - *isvalid* – contains 1 if the i-node is in use and 0 if it is free
  - *size* – length of the file in bytes
  - *direct* is a vector with the block numbers of the data blocks occupied by the file; the vector has 14 elements and each element is an unsigned integer occupying four bytes; these four bytes contain the value of the disk block number

- Blocks starting at *ninodeblocks+1* are used for storing the file's contents.

Please note the following simplifying assumptions:

- There is no bit map in the disk indicating if a block is free or occupied; this information is in the i-node table. When one mounts a disk, an array of unsigned chars is built in memory, indicating if a given block is free or occupied
- Files do not have symbolic names being identified by its i-node number; according to this, the file system does not contain directories
- When a file is created, it is associated with the first free i-node (F). After this, the file is known through the internal name F

Phase one – Implementation of the file system operations (65 % of the grade)

1. Disk emulation

The disk is emulated in a file and it is only possible to read or write data in 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***, which **is complete for this first phase** (i.e. you do not have to change anything). 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.

2. File system operations

The file *fs.h* describes the public operations to manipulate the file system. File ***fs.c*** contains the implementation of those functions and **is incomplete**.

`void fs_debug()` – prints detailed information about the file system (see details below)

`int fs_format()` – formats the disk

`int fs_mount()` – mounts the filesystem (reads the superblock and the i-node table; builds the block map)

`int fs_create()` – creates a new file; returns the i-node number

`int fs_delete( int inumber )` – deletes the file with inode *inumber*

`int fs_getsize( int inumber )` – returns the size of the file *inumber*

`int fs_read( int inode, char *data, int length, int offset )` – reads *length* bytes, starting at *offset*, from file *inode*, and transfers the bytes to a buffer that starts on address *data*

`int fs_write( int inode, char *data, int length, int offset )` – writes *length* bytes, starting at *offset*, into file *inode* by transferring the bytes from a buffer that starts in *data*

A more detailed description of the operations follows:

fs_debug – reports the contents of the i-node table. An example of `fs_debug` output is:

```
superblock:
      magic number is valid
      1010 blocks on disk
      101 blocks for inodes
      6464 inodes total
inode 3:
      size: 4500 bytes
      blocks: 103 194
inode 5:
      size 8190 bytes
      blocks: 105 109
```

fs_format – Creates a new file system (FS) in the disk, destroying all the disk´s content. Reserves 10% of the blocks for the i-node table; this table is initialized with all the i-nodes (non-valid). Please note that formatting a disk does not imply that the disk is accessible. This is performed by the mount operation. Trying to format a mounted disk is not allowed; invoking *fs_format* with the disk in use should do nothing and return an error.

fs_mount – Verifies if there is a valid FS in the disk. If the FS in the disk is valid, this operation reads the superblock and, using the i-node table in disk, builds in RAM the map of free/occupied blocks. Please note that all the following operations should fail if the disk is not mounted.

fs_create – Marks the first free i-node as occupied by a file of length 0. Returns the number of the allocated i-node. In error, returns -1.

fs_delete(int inumber) – Removes the file with i-node *inumber*. Frees the i-node entry and declares all the blocks associated with it as free by updating the map of free/occupied blocks. Returns 0 if success; -1 if an error occurs.

fs_getsize(int inumber) – Returns the length of the file associated with the i-node. In error, returns **-1**.

fs_read(int inode, char *data, int length, int offset) – Transfers data from a file (identified by a valid i-node) to memory. Copies *length* bytes from the i-node *inode* to the address *data* pointer, starting at *offset* in the file. Returns the effective number of bytes read. This number of bytes read can be lower than the number of bytes requested if the distance from *offset* to the end of the file is less than *length*. In case of error, returns -1.

fs_write(int inode, char *data, int length, int offset) - Transfers data between memory and the file designated by *inode*. Copies *length* bytes from the address *data* to the file starting at position defined in *offset*. This operation will allocate the necessary disk blocks. Returns the number of bytes really written to the file; this number of written bytes can be lower than the length, ion case there are no free disk blocks. In case of other errors, returns -1.

3. A shell to manipulate the file system

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

```
$ ./sf-1920 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
    create
    delete <inode of file in FS1920>
    cat     <inode of file in FS1920>
    copyin  <name of file_in the local file system> <inode of file in FS1920>
    copyout <inode of file in FS1920> <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. 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* executes 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 3
```

4. Implementation details

The following text presents some details of the available source code.

The following constants result form the description in 1.

```
#define DISK_BLOCK_SIZE    4096
#define FS_MAGIC           0xf0f03410
#define INODES_PER_BLOCK   DISK_BLOCK_SIZE / 64    // sizeof(struct fs_inode)
#define POINTERS_PER_INODE 14
```

The superblock and the i-node table correspond to:

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

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

How to handle the distinct contents of a disk block with 4 Kbytes:

- SuperBlock: contains a *struct fs_superblock*
- Block with i-nodes: contains an array of i-nodes `struct fs_inode inodes[POINTERS_PER_INODE]`
- Data block: corresponds to `char data[DISK_BLOCK_SIZE]`

Remember that the second argument of *disk_read* is an array of DISK_BLOCK_SIZE bytes. How to read and write the superblock and a block with i-nodes? Using the `union` data type of C language. A `union` is like a `struct`, but the members of the union all share the same memory space:

```
union fs_block {
    struct fs_superblock super;
    struct fs_inode      inodes[INODES_PER_BLOCK];
    char                 data[DISK_BLOCK_SIZE];
};
```

The size of the union `fs_block` will be always 4KB, i.e. the size of the largest element in the structure. The following code fragment illustrates the use of `union fs_block`

```
union fs_block block;
disk_read(0,block.data); // Uses disk_read to read  block 0 in an unstructured way
x = block.super.magic; // interprets the data read as the contents of a super block
disk_read(21,block.data);
x = block.inodes[2].size; // interprets the data read as an array of i-nodes
```

5. Work to do

Download the *mini-projeto.zip* archive file from CLIP. The zip contains:

- *Makefile*
- *disk.c* and *disk.h* : implements the operations in section 2
- *fs.h* and *fs.c* : implements the functionalities described in section 3. *fs.c* is incomplete and is the only one you should modify in this first part of the assignment
- *shell.c*: as described in section 4
- one or more files with disk images in FS1920 format; these type of files are identified by its name ending in .XY indicating a pseudo-disk with 10*X+Y blocks

The following figure illustrates the functionality contained in each file:

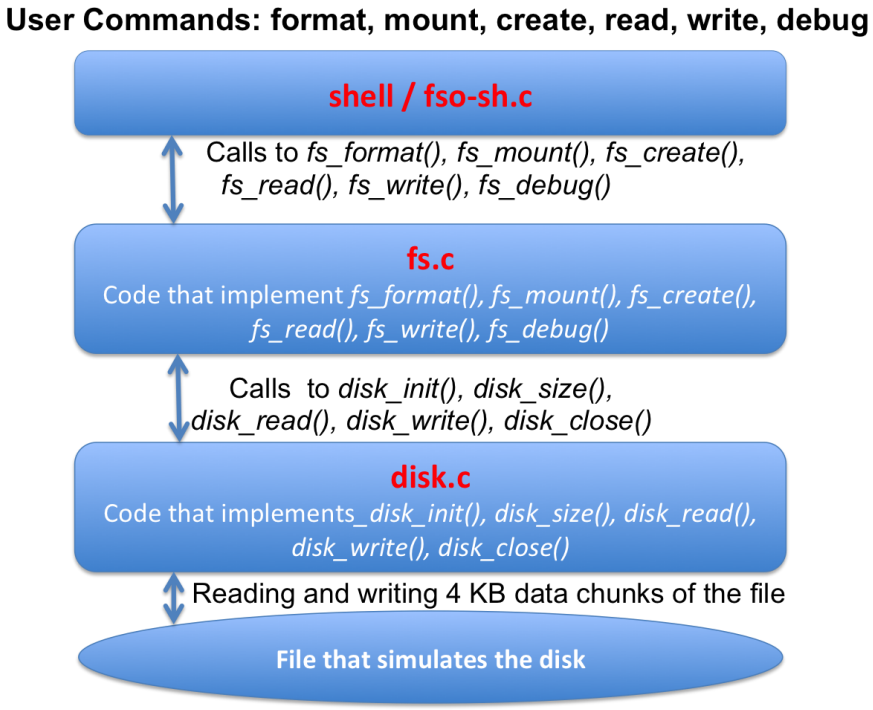

Figure 2

Complete the missing parts of **fs.c** in order to fulfill the functionalities described.

6. Recommendations

Develop the functions in the order below: (some of them will be already available in fs.c)

1. Start with *debug, format* and *mount*. Test your code using the supplied disk images
2. Implement the creation and destruction of i-nodes, ignoring the file contents. Test your code using the supplied disk images
3. Implement the file reading (*fs_read*)
4. The last operation to implement should be *fs_write*.

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

- **Do not forget to handle error situations**: your code must deal with situations like a file too big, 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).

- **Identify common functions:** access to the i-node *inumber* will be used in several situations. Accordingly, it will be useful to define the following two functions to read and write the contents of a i-node *inumber* file

```
void inode_load( int inumber, struct fs_inode *inode ) {
// reading contents of i-node inumber to an empty struct fs_inode
 ...
}
void inode_save( int inumber, struct fs_inode *inode ) {
// updating contents of i-node inumber from the contents of a previously filled
// struct fs_inod
...
}
```

- **Do not optimize your code:** as you may verify during the execution of your work, a simple shell command may correspond to hundreds of operations on disk. Try to justify why this happens but do not optimize your code.

- **How to prepare a new empty disk:** invoke the fso-fs with a non-existing file

  ```
  $ ./sf-1920 filename size
  ```

  Namely, 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 as in the following:

  ```
  $ ./sf-1920 filename
  ```

Phase two – Implementation of a disk block cache (35 % of the grade)

The goal of this second phase is to implement a **disk block cache (only) for data blocks**, which allows a better performance on accessing the files' data both for read and write operations. For this it is necessary that you modify the code from phase one as specified in the following.

1. Modifications to the disk emulation code

The cache for disk data blocks is implemented at the disk emulation level and has the following properties:

- It has a specified number of blocks (***cache_nblocks***) where to some disk blocks are to be cached; that number is *20%* of the total number of blocks in the disk, i.e. this number must be *nblocks*0.2* rounded to the nearest upper integer
- The block replacement algorithm is *random*, meaning that a block is randomly chosen to be evicted
- The write policy is *delayed write* (or *write back*) meaning that the modified blocks are only written to disk when they are evicted from the cache; a cache *block B* is written to disk in three situations:
    - when the file it belongs to is closed
    - when the cache is full and *B* is randomly selected as a victim for replacement, its contents have to be written on disk to make place for a new block
    - when there is an explicit call to a flush operation

To support these properties, you have to specify in *disk.c* the data structures that describe the cache, in the following way:

```
typedef struct __cache_entry {
      int disk_block_number;   // identifies the block number in disk or -1 for a free entry
      unsigned int dirty_bit;  // this value is 1 if the block has been written, 0 otherwise
      char *datab;     // a pointer to a disk data block cached in memory
} cache_entry;

cache_entry *cache;
```

To reserve space for the total cache blocks (***cache_nblocks)***, you have to use the *malloc* function to dynamically allocate the necessary memory both for all cache entries and for the data blocks in the cache. Each block with length `DISK_BLOCK_SIZE` in the cache points to a region in this latter allocated memory via the pointer ***datab*.**

Modifications to the disk emulation API

The implementation of the disk cache for data blocks require the following modifications:

`void disk_init( const char *filename, int nblocks )` – this function has to be **updated** to include the initialization of the disk block cache as specified above

`int disk_size()` – this function does not need to be modified

`void disk_read( int blocknum, char *data )` – this function is the same as the one in the first part of this assignment and corresponds to a synchronous read of a block on disk. This function is **not be modified**, i.e. ***it does not use the cache*** and continues to be used at the file system level to read the metadata on disk (e.g. superblock and i-nodes)

`void disk_read_data( int blocknum, char *data )` – this is a ***new function*** that **uses the cache** whenever a *data block* has to be read from disk

`void disk_write( int blocknum, char *data )` – this function is the same as the one in the first part of this assignment and corresponds to a synchronous write of a block on disk. This function is **not be modified**, i.e. ***it does not use the cache*** and continues to be used at the file system level to write the metadata on disk (e.g. superblock and i-nodes)

`void disk_write_data()` – this is a ***new function*** that **uses the cache** whenever a *data block* has to be written on disk.

`void disk_flush ()` – this is a ***new function*** that flushes *all the dirty data blocks* in the cache onto disk

`void disk_close ()` – this function has to be **updated** to guarantee that all *the dirty data blocks* in the cache have been/are updated on disk, before the disk is closed.

The semantics of the new read and write operations are described in following pseudo-code. Please be aware that you have to identify possible error situations in your code.

```
void disk_read_data( int blocknum, char *data ) {
	searches (the cache entry) for the block blocknum in cache
	if not present
		selects an entry in the cache to contain the new disk block
			// i.e. searches for a free block or for a block to be evicted
			// in the latter, guarantees that a dirty block is updated on disk
		updates the cache entry with the new blocknumber and puts dirty_bit to 0
		reads the block number from disk into the address datab in the cache entry
	copies the data in the data block in the cache, from the address datab into
				the address data received as argument
}


void disk_write_data( int blocknum, char *data ) {
	searches (the cache entry) for the block blocknum in cache
	if not present
		selects an entry in the cache to contain the new disk block
			// similar to the code in the disk_read_data
		updates the cache entry with the new blocknumber
	copies the data block from the address data into the address datab in the cache entry
	sets its dirty_bit to 1
}
```

2. Modifications to the file system API

To guarantee that your file system implementation is aware of a disk block cache, you have to implement the following modifications

`int fs_read( int inode, char *data, int length, int offset )` – this function has to be modified to use the new operation `disk_read_data()` whenever a file's data block is to be read from the disk. The reading of the meta-data is unchanged, i.e. the read is performed with the available operation `disk_read()`

`int fs_write( int inode, const char *data, int length, int offset )` – this function has to be modified to use the new operation `disk_write_data()` whenever a file's data block is to be written onto the disk. The writing of the meta-data is unchanged, i.e. the write is performed with the available operation `disk_write()`

`int fs_close( int inode )` – this is a *new function* that has to guarantee that all dirty blocks of the current file in the cache have to be updated on disk

`int fs_flush( int inode )` – this function updates the file's dirty blocks in cache onto disk

3. Modifications to the shell that operates the file system

To debug your disk cache implementation it may be valuable to have an extra shell command:

`diskflush` – this command forces the update of all data blocks onto disk

`cat, copyin, copyout` – have to invoke fs_close() to force the update of all written cached data blocks onto disk

Bibliography

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