FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

DEPARTAMENTO DE INFORMÁTICA

The BFS File System

(Mini-Project, due 22 December 2020, 23:59)

Please read this document carefully.
Your productivity (and maybe even success) strongly depends on the attention you gave it.

1. Objective

The goal of this assignment is to complete a "file system dump" operation of a simple file system, which is inspired by some UNIX-like file systems (MinixFS and UFS). The Basic File System (BFS) does not use the full features of the abovementioned FSs, like subdirectories or permission checking. However, space is reserved for a single, fixed-size, directory.

The next sections are organized in **three** parts.

1. The first part defines the BFS "on-disk" format.
2. The second part is the assignment itself and describes the task you must complete in the **mandatory** part, which is dumping the basic information of a BFS-formatted disk with some files in it (successful completion awards you a 55% of the grade).
3. The **optional** third part a) awards you an extra 20% grade if you are able to dump the contents for each file; and b) the last 25% are reserved for those who are able to develop an implementation of an integrity verification (a.k.a., fsck – file system checker) that reports any inconsistencies in the BFS disk.

2. The BFS file system

The BFS (Basic FS) file system is stored in a *disk* simulated by a *file*. It is possible to read and write *disk blocks* to this simulated disk, corresponding to *fixed size data chunks*.

A disk formatted with the *BFS format* has the following organization (see Section 2.2 for further details):

- A *super-block* describing the disk's organization
- A bytemap to manage inodes
- The next ***I*** blocks contain the i-node table, where the used i-nodes contain metadata on existing files in the FS. This table occupies 3% of the disk data blocks (rounded up)
- The first disk block after the i-node area is reserved for the directory (even if not implemented)
- The block after the directory is a bytemap to control the allocation of data blocks
- Finally, the remaining disk blocks are used for the files' contents

Figure 1 shows an overview of a disk with **40** *blocks* in total, after being initialized with the *format* command.

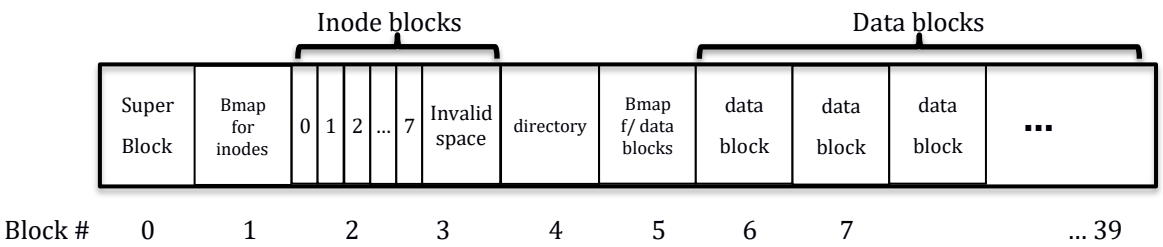

Figure 1

2.1. BFS layers

Figure 2 depicts the BFS abstract layers; if it was made available to you (it is too complex for the time we have now for this assignments), you would see that each one is implemented with a set of "independent" C files, usually just two: a file ending in `.c` and another in `.h`.

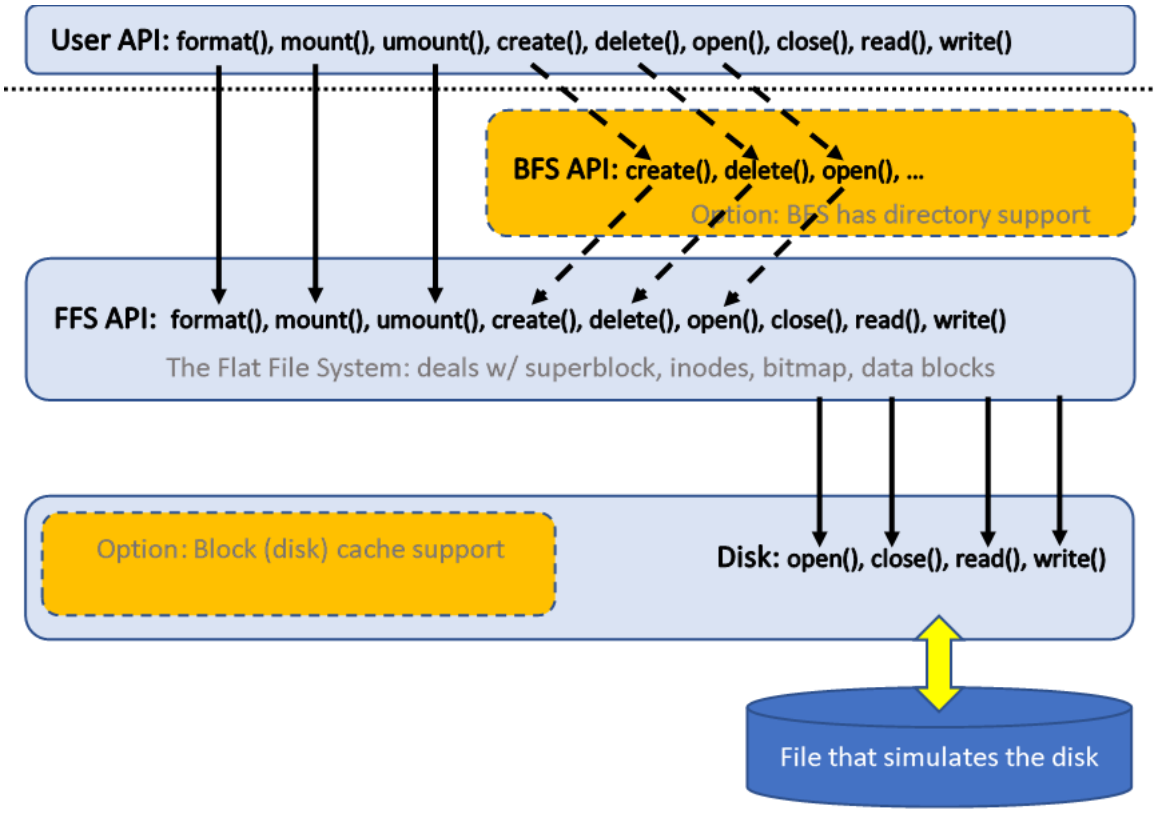

Figure 2

For the purpose at hand, the supplied files follow an identical structure and naming as used in TPC3 (now with a `ffs_` prefix), i.e., they are `ffs_bytemap.{c,h}`, `ffs_inode.{c,h}` and `ffs_disk_driver.{c,h}`; and two new files for the superblock code: `ffs_super.{c,h}`.

Notice that some functions are available in the user-level API, i.e., above the dotted line, while others are not (e.g., those available at the Disk layer). Also, notice that if there was support for a directory, some functions, e.g., `create()` would indirectly call functions in other modules before landing on the most important module, the BFS FFS. As with TPC3, you will deploy an object-oriented programming style in C, and have functions that look like they have the same name in various "places".

2.2. The on-disk BFS layout

The *BFS on-disk organization* is described (see also Figure 1) in the following points:

- 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* (see the `struct superblock` later in the document). The "base" information is:
    - *fsmagic* "magic number" (**`0xf0f03410`**) - used to verify if the file image is a BFS formatted disk
    - *nblocks* - total number of blocks (defines the disk size, in blocks)
    - *nbmapblocksinodes* - number of blocks reserved to store the bytemap for the inodes (always 1 in BFS)
    - *ninodeblocks* - number of blocks reserved to store i-nodes
    - *ninodes* - total number of i-nodes
    - *nbmapblocksdata* - number of blocks reserved to store the bytemap for data blocks (always 1 in BFS)
    - *ndatablocks* - number of "user-available" data blocks

However, some more pieces of information have been added, to simplify the access to the different structures in the disk. These are "redundant" in the sense that they can be computed from the (above) previous data items, but having it already computed is a bonus.

- *startInArea* – disk block (absolute address) where the inode area starts – i.e., i-node table offset
- *startDtBmap* – disk block (absolute address) where the data bytemap area starts
- *startDtArea* – disk block (absolute address) where the data area starts
- *mounted* – indicates if the disk is mounted, or was left mounted (failure or bug)

- Blocks starting at 2 and finishing in *ninodeblocks* contain i-nodes. Each **i-node in the i-node 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* – indicates the length of the **file** in bytes
  - *direc*t is a vector with the block numbers of the data blocks occupied by the file; the vector has 6 elements, and each element is an unsigned integer occupying four bytes; these four bytes contain the number (a.k.a. address) of the disk block that element "points to".
- The block starting at *ninodeblocks +1* is used to store the "1-block-sized" BFS directory (even if not used)
- The block at *ninodeblocks +2* is used to store the bytemap that manages data blocks (only 1 is used)
- Blocks starting at *ninodeblocks + 2 + nbmapblocks* are used for storing the files' contents (i.e., user data).

2.3. Other BFS structures: bytemaps and i-nodes

There other structures are the bytemaps (two of them) and i-nodes (grouped in "a table").

As you know, from TPC3, bytemaps are arrays (or blocks, if you prefer) of bytes; in BFS, a bytemap is stored in a disk block, but not all entries are significant – only the first *ninodes* entries, for the bytemap that is used to manage i-nodes, and only the first *ndatablocks* entries, for the bytemap that is used to manage data blocks.

The same happens with i-nodes: they are stored in *ninodeblocks* blocks, each block holding **INODES_PER_BLOCK** i-node structures (see below), but in the "last" block not all i-node entries may be significant – the total number of i-nodes is *ninodes*.

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

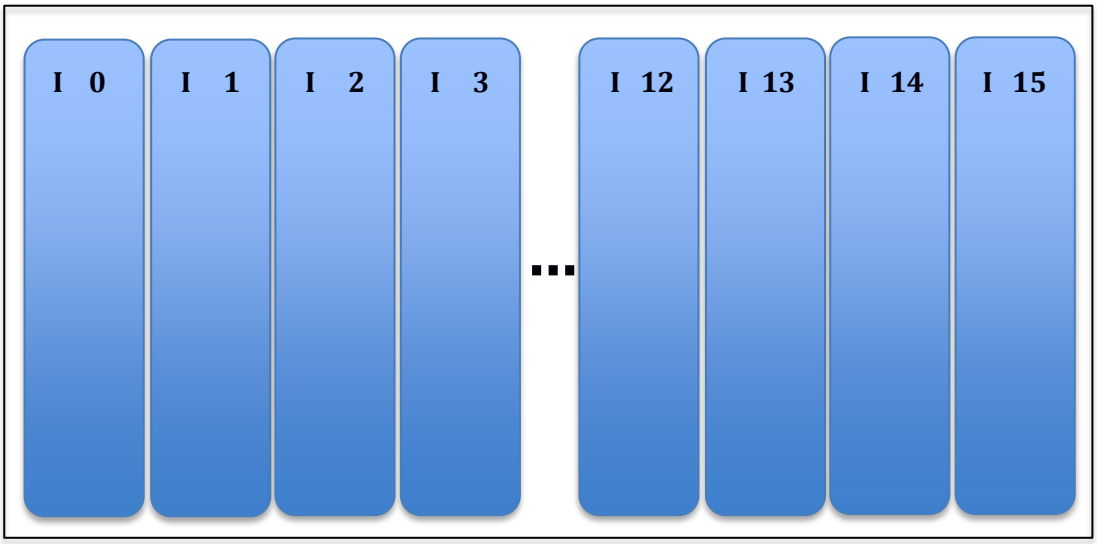

Figure 3: i-node structure (above) and 16 i-nodes packed in a block

Mandatory

Implementation of the basic file dump system operation (55 % of the grade)

1. Disk driver operations (full implementation provided)

The disk is emulated in a file and it is only possible to read or write data in chunks of 512 bytes; the file `disk_driver.h` defines the API for using the virtual disk. For further reference, consult the TPC3 document.

2. Implementation of the FFS operations (all required)

Please note the following simplifying assumptions:

- Files do not have symbolic names, a file is identified by its i-node number; according to this, the file system does not contain (for the moment) directories
- When a file is created, it is associated with the first free i-node (which is identified by a number, N), all the fields are cleared (set to zeroes) except *isvalid*, which is set to 1. After this, the file is known through the internal name N. As things progress, files (i.e., i-nodes) located before may be deleted, and thus be invalid.
- Below, to shorten the API description we use **`uint`** as an abbreviation of **`unsigned int`**, but in the files provided to you we use the full spelling of the words, i.e., **`unsigned int`**.

BFS operations can be further subdivided into two groups: those acting on the on-disk file system, and those that deal with in-memory data structures – which we are not interested in, on this assignment.

2.1. A short and quick guide to the implementation

Delivery:

- You must deliver all the `.h` and `.c` files, **except** `disk_driver.{c, h}`, in a zip to Mooshak
- You must implement a main program in file `dumpBFS.c` and, if you want/need it, add stuff to file `dumpBFS.h`. The `dumpBFS` program code handles **both** the **Mandatory** part **and Option A** (if implemented) parts
- You should (but it is just a suggestion!) implement operations for different structures on different files (you will get the ones listed below with the main structures already declared):
    - superblock operations on `ffs_super.{c,h}`
    - i-node operations on `ffs_inode.{c,h}`
    - bytemap operations on `ffs_bytemap.{c,h}`

Program output:

What follows is (a snippet, corresponding only to the superblock information) of an example of program execution and output for user debugging (**not for Mooshak!**):

```
$./dumpBFS disk0
Superblock:
fsmagic                 0xf0f03410
nblocks                 100
nbmapblocksinodes       1
ninodeblocks            2
ninodes                 18
nbmapblocksdata         1
ndatablocks             94
startInArea             2
startDtBmap             5
startDtArea             6
mounted                 0
```

... more lines, not shown ...

The same execution, but now intended for Mooshak, would print (notice that some parts have a different format, marked in yellow – if the magic number is incorrect, it should print **`invalid`**; if the disk was left mounted, print **`yes`**)

```
$./dumpBFS disk0
Superblock:
valid
100
1
2
18
1
94
2
5
6
no
```

... more lines, not shown ...

What follows is the i-node bytemap, and the output should follow the same format that you used on TPC3, except for the addition of a header:

```
Bmap for i-nodes:
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
```

Now, the i-node table:

```
i-nodes:
0:
        valid
        1234
                0
                1
                2
                NULL
                NULL
                NULL
1:
        invalid
2:
        valid
        123
                5
                NULL
                NULL
                NULL
                NULL
                NULL
3:
        Invalid
```

(… here we omit all the other invalid i-nodes from 4 to 16, but you must print all entries as invalid)

```
17:
        invalid
```

Now, the bytemap for data block management:

```
Bmap for data blocks:
1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0
```

Before addressing the final – **but optional** – printing issue, which is printing the contents of the files (i.e., the files' data blocks) we want you to pay attention to the following: the addresses (numbers) in the bytemaps and in the i-node pointers are relative to the start of the target zone, e.g., if the "Bmap for data blocks" says that, for some file, block 0 is valid (in-use), that means the following – in disk absolute addresses, that block's address is 6 (remember: the superblock entry **`startDtArea`** states that the data block area starts at physical block 6, so the file block 0 is physical block 0 + 6 → 6).

2. Some hits for the implementation

What follows is a number of suggestions, including a high-level algorithm; you may use the suggestions, the files supplied .h and .c files, or not… In *extremis*, you may deliver only the `dumpBFS.c` (using the `dumpBFS` files, of course).

dumpBFS:

declare one variable of type struct super sb;
declare two blocks/arrays of 512 bytes, one bmap_inodes, the other bmap_dataBlocks
super_ops.read(&sb) will fill the super variable
print the information 1

bytemap_ops.read(bmap_inodes)
print the information 2

for (int i= 0; i < number of inodes; i++)
    inode_ops.print_inode(i) here you may have a 2nd parameter for DEBUG or not

bytemap_ops.read(bmap_dataBlocks)
print the information 2

THE END – mandatory part

More suggestions:

print the information 1	you must decide if that will be a main program function, or if you prefer to implement it as a super_ops.print()
print the information 2	the same as above: main program or OO function? You decide… However, remember that there is no difference (other that where the data is located on-disk between the two bytemaps, so one function is enough…but then it needs to access two distinct maps. A parameter?
On i-node numbers	Look at figure 3 (to the block with i-nodes inside). If you have two i-node blocks, and the one on figure 3 is the first block (of the i-node table), what is the number of the 1st i-node in the second block? Is it 0? Is it 16? Using consecutive numbers has the advantage of keeping the same number as the corresponding bytemap entry, i.e., bytemap entry number, e.g., 17, corresponds to i-node number 17. So, what is the disadvantage? This one: given an i-node number (e.g., 17) you will have to compute the i-node block where that entry is stored (that's i-node block 1) and the offset "inside" that block (and it is offset 2)

Finally, remember that there are logical block numbers, e.g., i-node block 1, and their physical location in the disk – in this case, block 3 – see figure 1: disk block 0 is the superblock, disk block 1 is the bytemap for the i-nodes, disk block 2 is the first (i.e., block 0) of the i-node blocks, so finally block 3 is the second i-node block (i.e., block 1).

In short, you need two functions (spelled here with very long names 😊).

void convert_inode_nbr_to_location(uint logical_inode_nbr, uint *logical_block, uint *offset)
int convert_inode_logical_block_to_physical(uint logical_block) // returns the physical

3. Testing your implementation

You will get, in a zip posted on CLIP, along with the includes and "skeletons" for the `.c` files, different "disks" – a blank disk, two formatted (with 4 and 18 i-nodes) but without any files on them, and several disks (all with 100 blocks of 512-bytes per block) formatted with 18-inodes, most with only 1 file, but of different (file) sizes, etc.; you will also get text files with the output produced by the instructor's implementation of `dumpBFS` and that output shows you the definitive format (in case there is any difference with the samples previously presented in pages 4 and 5).

Expect at least the following files for your tests (all disks are 100 512-byte blocks long and all disks with files have 18 entries):

Description	Filename	Output Listing
Unformatted disk	disk0.unformatted	disk0.unformatted.out
Formatted, 4-entries, no files	disk0.4entries.empty	disk0.4entries.empty.out
Formatted, 18-entries, no files	disk0.18entries.empty	disk0.18entries.empty.out
F., 1 file 512B with letters A	disk0.1F.512B	disk0.1F.512B.out
F., 1 file 512B ½ letter A, ½ B	disk0.1F.512B.A+B	disk0.1F.512B.A+B.out
F., 1 file, a copy of "errno.h"	disk0.1F.1497B	disk0.1F.1497B.out
F., 2 files, "errno.h" and As	disk0.2F.1497B+64B	disk0.2F.1497B+64B.out
F., 2 files, both with size 0	disk0.2F.0B	disk0.2F.0B.out

Notice that the text "Printing contents of file(inode) …" (where the i-node number is printed) is only shown as a result of implementing Option A.
If you implement Option A, you should submit **both the Mandatory part** (say, for example, to Mooshak contest FSO2021-Mini-M) **and Option A** (say, for example, to Mooshak contest FSO2021-Mini-A) to get a) both grades and b) if you fail on Option A and have not submitted the Mandatory part, you will end up with a very low grade (maybe even zero ☹),

Option A

Implementation of the dump of the file contents (20 % of the grade)

1. Objective

If you implement this option, the contents of every file must be printed as characters (files in the disks we supply are text files of different sizes), e.g., you should use something similar to **`putchar(c)`** or **`printf("%c",c)`**, and you should change line only when the sixteenth character has been printed – something that may look like

```
File 0 data blocks:
Era uma vez um d
ump de um disco0
... ... ... ...
e acabou aqui
```

You should note that, as the file's text will probably have new-line characters itself, a new-line may appear before the 16th character has been reached (or, to make things even funnier, it may appear exactly at the 16th character, and there will be what seems to be an "extra" blank line), and the output will look "broken", while it is, in fact, correct:

```
File 0 data blocks:
Era uma vez← a new-line character was here
um d← this is the 16th character, so we inserted a new-line 😊
ump de ...
... ... ... ...
```

All the file's data blocks must be printed out sequentially, without spaces, and you must stop at the file's last valid character (as dictated by the file's length), i.e., you should not print "garbage" beyond the end-of-file 😊

2. Some hits for the implementation

If you implement this option, you need to access the i-node data structure. Therefore, it may make sense (or not…) to implement it as an operation on i-node structures. If you adhere to the OO-style, maybe it could be

```
inode_ops.printdata(inodenumber)
```

where it would fetch and print all the data of the i-node whose number is specified in the argument, up to the file length, which is specified in the i-node's size field… if the i-node is valid; otherwise, it would just print **`invalid`** (and the i-node number, of course).

Notice that it confuses people if you, for similar tasks, take different approaches, sometimes using a "simple" function with access to global data and then, on other situations, use an OO-style…

3. Formats

Look at the .out files; as you can see, you only print file data for files (i-nodes) who do exist and have data (as an example, if i-node 2 is valid but with size 0, the output is similar to the following

```
Printing contents of file(inode) 2

** NO DATA **
```

4. Delivery

As this is just an extension to dumpBFS, this delivery is through Mooshak (a different contest will be created)

Option B

Implementation of integrity checking of BFS (25 % of the grade)

Delivery:

- **Delivery is by e-mail** (be aware that your submission will be evaluated **by the instructors**, not Mooshak)
- You must deliver all the `.h` and `.c` files, and a Makefile in a zip, **except** `disk_driver.{c, h}`.
- You must implement a main program in file `fsckBFS.c` and, f you want/need it, in file `fsckBFS.h`.
- The program is executed with just one argument, the disk name (e.g., **`$./fsckBFS disk0`**)
- You should implement operations for different structures on different files (you should use the same files used in the mandatory part, and, if needed, you can extend them with new structures and/or code):
  - superblock operations on `ffs_super.{c, h}`
  - i-node operations on `ffs_inode.{c, h}`
  - bytemap operations on `ffs_bytemap.{c, h}`

If you implement this option, you should check the consistency of everything 😊. Here are some examples of inconsistency situations:

- The obvious *magic number* 😊
- The size of the disk stored in the superblock is different from what is reported by the **`stat`** "command" of the disk driver
- The sum of the sizes of the different regions is different from the size of the disk
- There are entries in the bmap of i-nodes that contradict the `isvalid` of the corresponding i-nodes
- There are entries in the bmap of data blocks that contradict the information on the i-nodes (this one requires some work! 😊)
- …