FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

DEPARTAMENTO DE INFORMÁTICA

The BFS File System
(Mini-Project, due 29 December 2021, 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 continue the implementation 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 permission checking and multiple levels of subdirectories. It is an extension of the work previously developed for TPC3. 😊

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 extending the inode structure, and implementing a single-level of subdirectories (successful completion awards you a 120 points – or 12,0 valores).
3. The **optional** third part awards you up to an extra 80 points, depending on your implementation of **`file_read()`** and **`file_write()`**: if your functions only produce correct results with buffers of 512 bytes (i.e., the buffer is exactly the same size of the disk block), then you get an extra 20 points; however, if you are able to develop a generic implementation that works with a file buffer of any size, you get the maximum 80 points.

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 an even number of disk data blocks specified as an argument to the format command
- The first disk block after the i-node area is reserved for the directory
- 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.

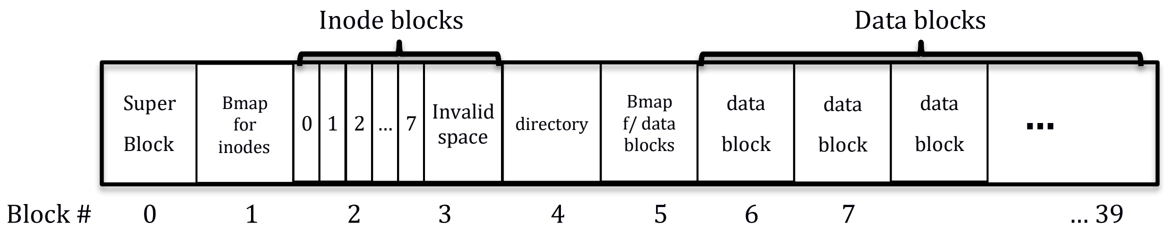

Figure 1

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`.

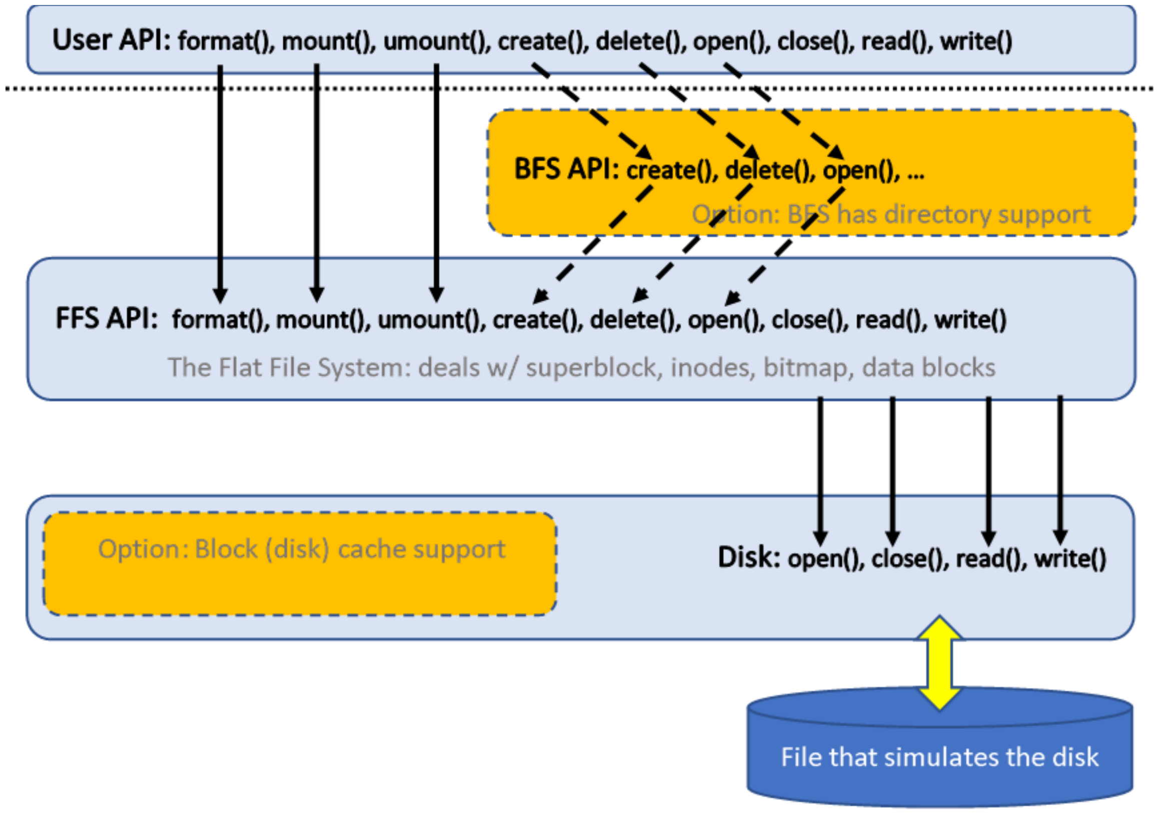
User API: format(), mount(), umount(), create(), delete(), open(), close(), read(), write()
BFS API: create(), delete(), open(), ...
Option: BFS has directory support
FFS API: format(), mount(), umount(), create(), delete(), open(), close(), read(), write()
The Flat File System: deals w/ superblock, inodes, bitmap, data blocks
Option: Block (disk) cache support
Disk: open(), close(), read(), write()
File that simulates the disk

Figure 2 (Note: **only some API functions will be implemented**)

For the purpose at hand, the supplied files follow an identical structure and naming as used in TPC3 (now with a `ffs_` or `bfs_`prefix), i.e., they are `ffs_disk_driver.{c,h}`, `ffs_bytemap.{c,h}` and `ffs_inode.{c,h}` ; and new files for the superblock code, `ffs_super.{c,h}` and directory `bfs_dir.{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 there is now support for a directory, so some functions, e.g., `create()` 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 (always an even number, explained later)
    - *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 (due to, e.g., power failure or bug)

- The block at 1 is used to store the bytemap that manages i-nodes (only 1 is used)
- The next *ninodeblocks*, starting at (and including) block 2, contain i-nodes. Each **i-node** in the **i-node table** has the following contents (full description down, in 2.3.1, and in **`ffs_inode.h`**):
  - contains metadata about the i-node (in use or free, plus other information)
  - *size* – indicates the length of the **file** in bytes
  - an area for data pointers that contains information about the block numbers of the data blocks occupied by the file; that information depends on whether the allocation is contiguous, or indexed.
- Following the i-nodes, a block is used to store the "1-block-sized" BFS directory
- The directory is followed by a block that stores the bytemap that manages data blocks (only 1 is used)
- The next **`ndatablocks`** are used for storing the files' contents (including the content of non-root directories).

The *BFS superblock* is as described below:

```
struct super {
  unsigned int fsmagic;
  unsigned int nblocks;
  unsigned int nbmapblocksinodes;       // Always 1 in BFS
  unsigned int ninodeblocks;
  unsigned int ninodes;
  unsigned int startInArea;
  unsigned int startRotdir;
  unsigned int startDtBmap;
  unsigned int nbmapblocksdata;         // Always 1 in BFS
  unsigned int startDtArea;
  unsigned int ndatablocks;
  unsigned int mounted;
};

union sb_block {
  super sb;
  unsigned char data[DISK_BLOCK_SIZE];
};

/* operations on superblock structures
  create: To be called by format(), thus requires NOT mounted as it
          overwrites in the in-memory SB structure.
    parameters:
     @in: disk size (in blocks); percentage of disk size to allocate for inodes
     @out: pointer to superblock structure

  read: read: reads in a SB from disk, overwrites the in-mem SB structure.
        Requires mounted
    parameters:
     @out: pointer to superblock structure
  ...        ...          ...
*/

struct super_operations {
  int (*create)(struct super *sb);
  int (*read)(struct super *sb);
  int (*write)(struct super *sb);
  ...        ...          ...
};
```

2.3. Other BFS structures: bytemaps, i-nodes and directories

The other structures are the bytemaps (two of them), i-nodes (grouped in "two tables"), and directories.

2.3.1 i-nodes

i-nodes are stored in *ninodeblocks* blocks, each block holding some number of i-node structures (see below), and care has been taken so that all i-node entries are significant – the total number of i-nodes is *ninodes*. However, BFS supports both contiguous and indexed allocation, so there are two distinct inode formats: contiguous files have 2 data pointers but indexed files have 6 data pointers (5 direct and 1 indirect).

```
struct smlInode {
  unsigned char  isvalid;
  unsigned char  type;
  unsigned short size;
  unsigned short start;
  unsigned short end;
};
```

```
#define DPOINTERS_PER_INODE 5
struct lrgInode {
  unsigned char  isvalid;
  unsigned char  type;
  unsigned short size;
  unsigned short dptr[DPOINTERS_PER_INODE];
  unsigned short iptr;
};
```

The purpose of the **isvalid** is well known, from TPC3; **type** will be a character **C** for contiguous files, **D** for subdirectories, and **I** for indexed files. Subdirectories are held in contiguous blocks, so **C** and **D** file allocation structures have the same format, **struct smlInode**.

Most functions work with both types, so we have created:

```
union sml_lrg {
  struct smlInode smlino;
  struct lrgInode lrgino;
};
```

And, finally, to be able to read/write blocks of small and large inodes to the disk,

```
union inode_block {
  struct smlInode smlino[SML_INOS_PER_BLK];
  struct lrgInode lrgino[LRG_INOS_PER_BLK];
  unsigned char data[DISK_BLOCK_SIZE];
};
```

As examples, i) a contiguous file is described by a 8-byte = 2-byte mode + 2 byte size + 4-byte i-node structure, and ii) an indexed file is described by a 16-byte = 2-byte mode + 2 byte size + 12-byte i-node structure. So, 1 disk block packs a maximum of, exactly 😉, 32 i-nodes of the "indexed file type" or 64 i-nodes of the "contiguous file type".

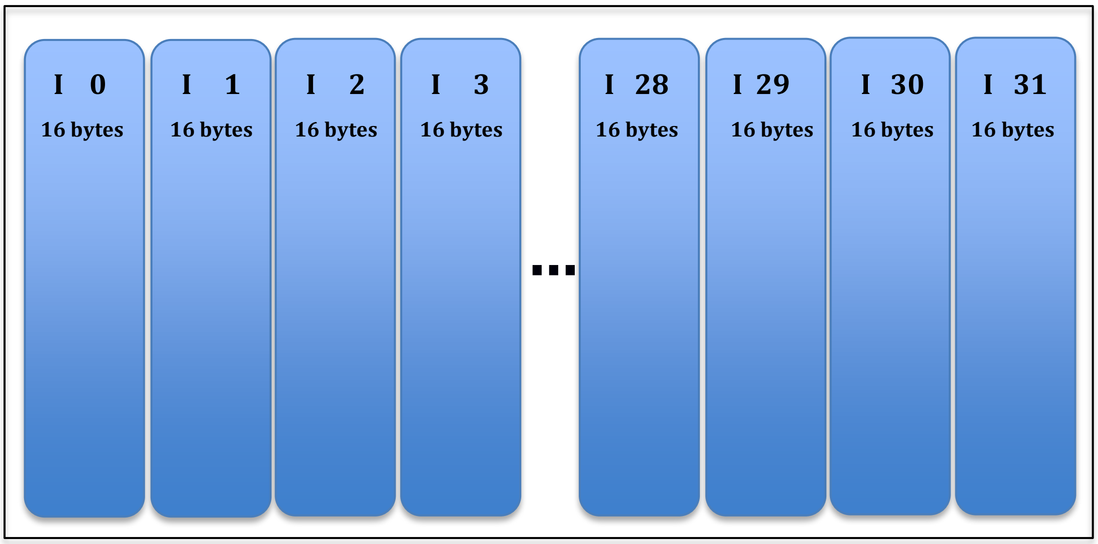

Figure 3: 32 large i-nodes adequate for indexed files, packed in a block

We have decided not to "mix" the two different-sized types in a single disk block, and that's why the number of inode table blocks is even: **the first** half is **for indexed** file allocation, **the 2nd** half **for contiguous** files (and directories, with a single block). So, the minimum number of inode blocks in a BFS disk is two.

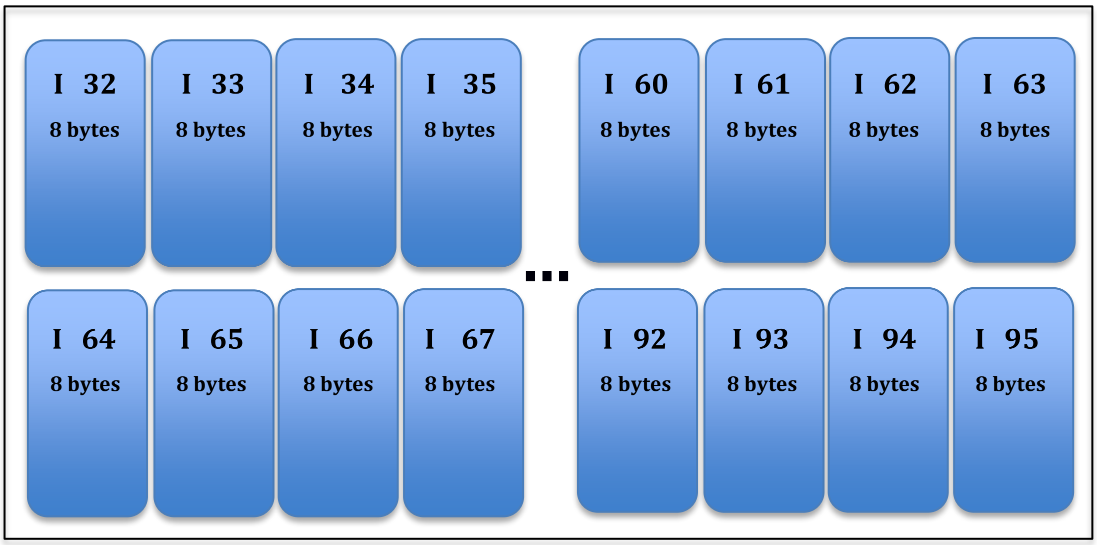

Figure 4: 64 small i-nodes adequate for contiguous files, packed in a block

Noticed that we have numbered the i-nodes sequentially, starting from zero. With such a scheme, it seems pretty easy to deduce which disk block to access for a particular i-node number (you do the math 😉). For the moment, **do not change this addressing scheme** (later on we will see if it is possible to lift this restriction, and anyone can "invent" its own addressing scheme)

2.3.2 Bytemaps

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.

In the case of BFS, the inode bytemap (1 block only) is logically split into two parts: **the 1st** part is used to **manage indexed** file allocation with **`struct lrgInode`**, **the 2nd** part **to manage contiguous** files using **`struct smlInode`**.

2.3.3 Directories

The BFS dentry structure is exactly the same as with TPC3. However, now, not only the same root directory block (as in TPC3) is used, but a single-level of subdirectories is supported. To create a subdirectory, a contiguous file with a single disk block is used (so, the directory contents reside in the data blocks area); the same functions (dir_ops, as defined in TPC3) are used, with a single difference: when a file is created, it's type, **`C`**, **`D`**, or **`I`**, must be specified.

3. Where to start?

3.1 Format a disk!

The first thing you must do is create the **`format()`** function, which will be called by the supplied main program, `testBFS.c` and fill-in the information in **`struct super`** and write it to block 0. The **`format()`** function i) has a parameter that specifies the total number of i-node blocks, and starts by calling disk_driver to get the size of the disk; ii) then, computes the number of blocks to use for the "two" i-node tables, and then computes all the remainder information to fill in **`struct super`**, and writes it to disk block 0. That's it 😊. But, for that, you must write the **`struct super_operations write()`**, then **`read()`**, and **`debug()`**, etc..

When all these functions are implemented, you can use `testBFS` to try them and see if they produce the required outputs…

DO NOT PROCEED to other implementations if **you have not finished & <u>tested</u>** the "format disk".

3.2 Files

Files are a new topic, one that we have not dealt with in TPC3. To illustrate how things are glued together in BFS, suppose that i) you want to create a contiguous file named FC, with a maximum size of 10 blocks, ii) open it for writing, and iii) after some writes you close it.

In BFS **create** is both a **directory** operation, one studied in TPC3, where we assign a name/inode for the file, and a "flat filesystem" operation, where we fill that inode with relevant information. For now, assume you have a few functions gathered together under

```
struct bfs_operations {
...
  int (*open)(char *filename);
  int (*close)(int fd);
  int (*create_C)(char *filename, unsigned int maxBlocks);
  int (*create_I)(char *filename);
  int (*delete)(char *filename);
  int (*write)(int fd, const void *buffer, unsigned int length);
  int (*read)(int fd, void *buffer, unsigned int length);
};
```

3.2.1 File creation

So, what we (the users) will do in our program is call `bfs_ops.create_C("FC",10);` that will need to:

a) call our good old `dir_ops.create("FC");`

b) call `in=bmap_ops.allocate(-1)`, to get the location of the 1st free i-node (**oops!** We now have 2 types of inodes – let's forget it, for now);

c) new operation: look into the data blocks byte map for 10 contiguous free entries, and allocate them;

d) store in the i-node `in` the address of the first and last entries returned on (c) above. The END.

3.2.2 File open

So, no we (the users) call `bfs_ops.open("FC");` that will need to:

a) call `dir_ops.???("FC")`. Hummm... we will need one more "public" directory operation, perhaps implemented with `dir_ops.readdir`, scanning the directory for that filename, and returning the dentry or dentry pointer, when found;

b) Now, with the dentry we have the i-node. Good. But we must also set the file pointer to zero, as we have just opened the file... (so our `bfs_file` module must have a private variable, say, `fp` – because we just open one file at a time). The END.

3.2.2 File write (and read)

So, no we (the users) call `ffs_ops.write(fd,buf,512);` that will need to:

a) TBD;

b) ...

c) ... The END.

Mandatory

Implementation of the basic file system operations (65 % of the grade)

1. Disk driver and bytemap operations (full implementations 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.

The bytemap operations as used in TPC3 are also provided in {`bfs_bytemap.h` and `bfs_bytemap.c`}. **However**, you **must** modify it because this project requires the manipulation of two independent bytemaps.

2. Implementation of the FFS operations (all required)

Please note the following simplifying assumptions:

- Files do have symbolic names, a file is identified by its unique name within a directory;
- 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" (i.e., with a i-node number less than N) 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 & deliverables

Delivery:

- You must deliver all the `.h` and `.c` files, **<u>except</u>** `disk_driver.{c, h}`, in a zip to Mooshak
- You will use the supplied main program, `testBFS.c` and, if you want/need it, you may add stuff to it and to the file `testBFS.h`. But Mooshak tests are performed with the original version; do not deliver the `testBFS.*` files. The `testBFS` program handles **both** the **Mandatory** part **and Optional** (if implemented) parts. In short, `testBFS.c` handles commands to
  - Create or re-create a disk
  - Format a disk
  - Dump information about on-disk BFS structures only (not contents of user files or directories)
  - Create, Delete and List information on Files and Subdirectories
  - Write and Read files
- 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}`
  - directory operations on `bfs_dir.{c,h}`
  - file operations on `bfs_file.{c,h}`

Program output: (THIS IS A PREVIEW. The correct output formats will be given later)

What follows is (a snippet, corresponding only to the superblock information) of an example of program execution:

```
$ ./tstBFS < testeC.txt
D disk0 100      OK
L                OK
F disk0 4        Formatting...
L                OK
Superblock:
  fsmagic           = 0xf0f03410
  nblocks           = 100
  nbmapblocksinodes= 1
  ninodeblocks      = 4
  ninodes           = 192
  nbmapblocksdata   = 1
  ndatablocks       = 92
  startInArea       = 2
  startRotdir       = 6
  startDtBmap       = 7
  startDtArea       = 8
  mounted           = no
M disk0 1        OK
o                OK
Printing the large inodes bytemap ----------
1  0 1  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Printing the small inodes bytemap ----------
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Printing the data blocks bytemap ----------
1  1  1  0  0  1  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0
```

Now, the i-node table will have a format similar to this one:

```
i-nodes:
0:
       valid
       I
       1234
              0
              1
              2
              NULL
              NULL
              NULL
1:
       invalid
2:
       valid
       I
       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
```

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 8 (remember: the superblock entry **`startDtArea`** states that the data block area starts at physical block 8, so the file block 0 is physical block 0 + 8 $\rightarrow$ 8).

2. Some hits for the implementation

TBD

3. Testing your implementation

You will get, in a zip posted on CLIP, along with the includes and "skeletons" for the `.h` and `.c` files, and a few "specfiles".

Optional Part

Implementation of the read/write of file contents **(up to 35 % of the grade)**

1. Objective

If you implement this option, the contents of every file must be written or printed as characters, 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

TBD

3. Formats

TBD

4. Delivery

As this is just an extension to **`tstBFS`**, delivery is through Mooshak. Further details TBD.