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, which includes a reserved area for a root directory.
2. The second part is the assignment itself and describes the task you must complete in the **mandatory** part: a) superblock; b) bytemaps; c) extending the traditional inode structure that supports indexed allocation with contiguous allocation (successful completion awards you a 130 points – or 13,0 valores). The tests are performed with the non-interacting commands described in page 8.
3. The **optional** third part awards you up to an extra 70 points, is about: a) for 20 points - implementing a **root directory** that allows you to create and list files (with both types of allocation strategies); and b) for an extra 50 points also implement subdirectories in the root (each with a single block) where you may create more files. The tests are performed with the commands described in page 9.

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.

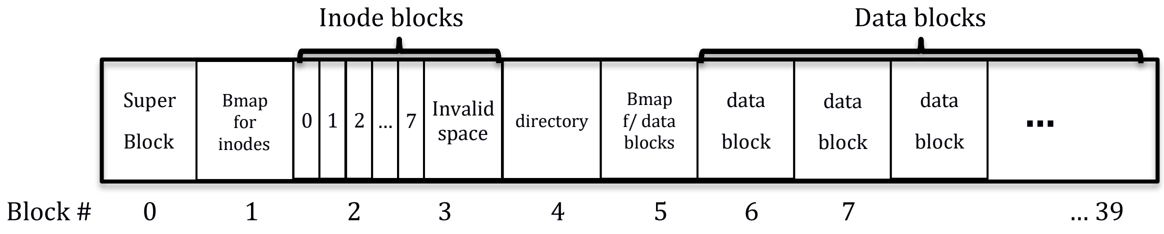

Figure 1

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`.

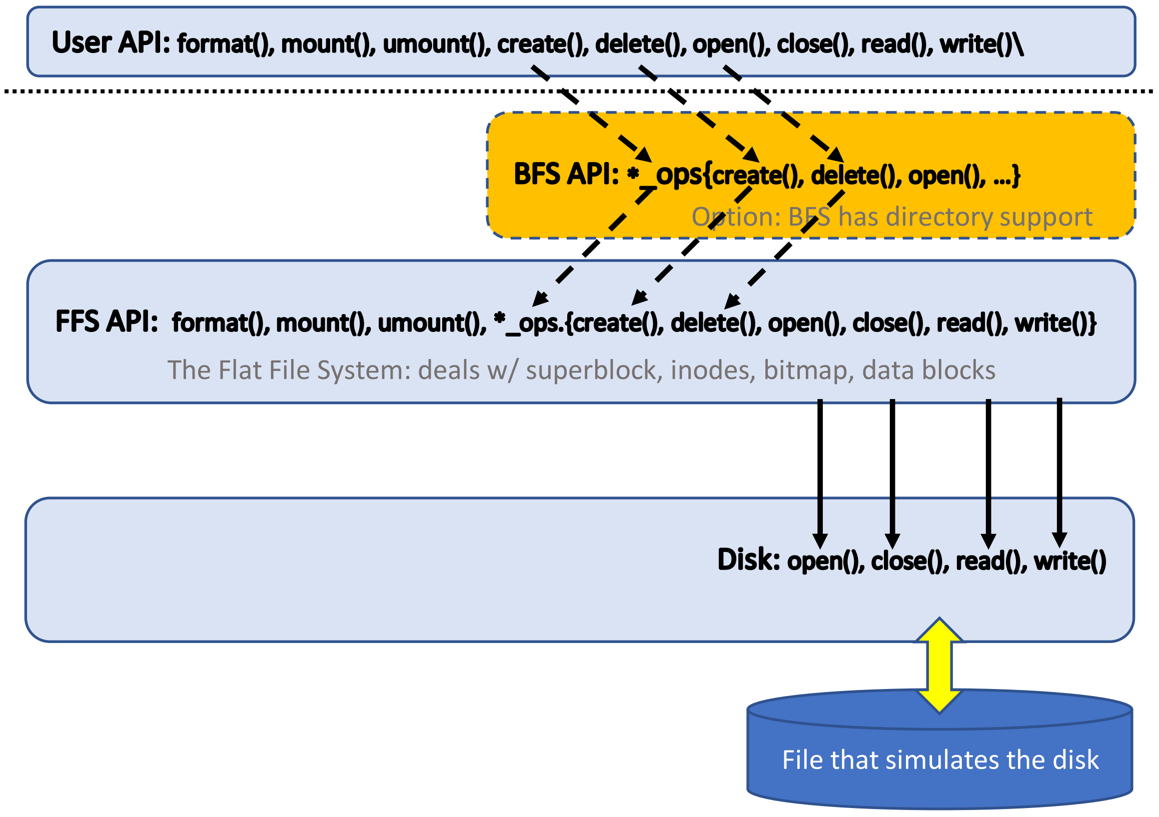
User API: format(), mount(), umount(), create(), delete(), open(), close(), read(), write()\
BFS API: *_ops{create(), delete(), open(), …}
Option: BFS has directory support
FFS API: format(), mount(), umount(), *_ops.{create(), delete(), open(), close(), read(), write()}
The Flat File System: deals w/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

Figure 2 (Note: **not all API functions shown**, and **only a subset 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".

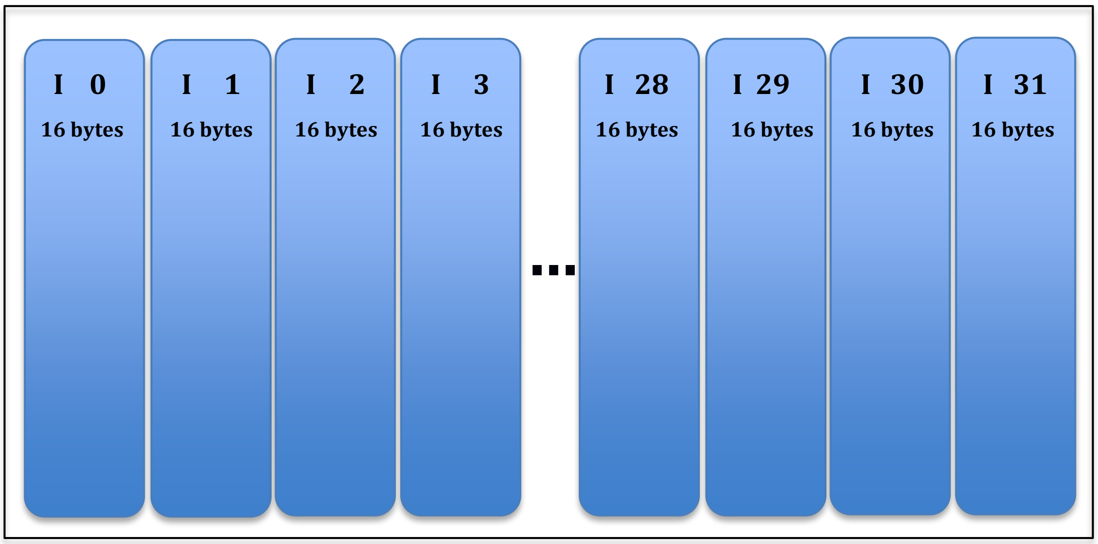

Figure 3: 32 large i-nodes adequate for indexed files, packed in a block

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.

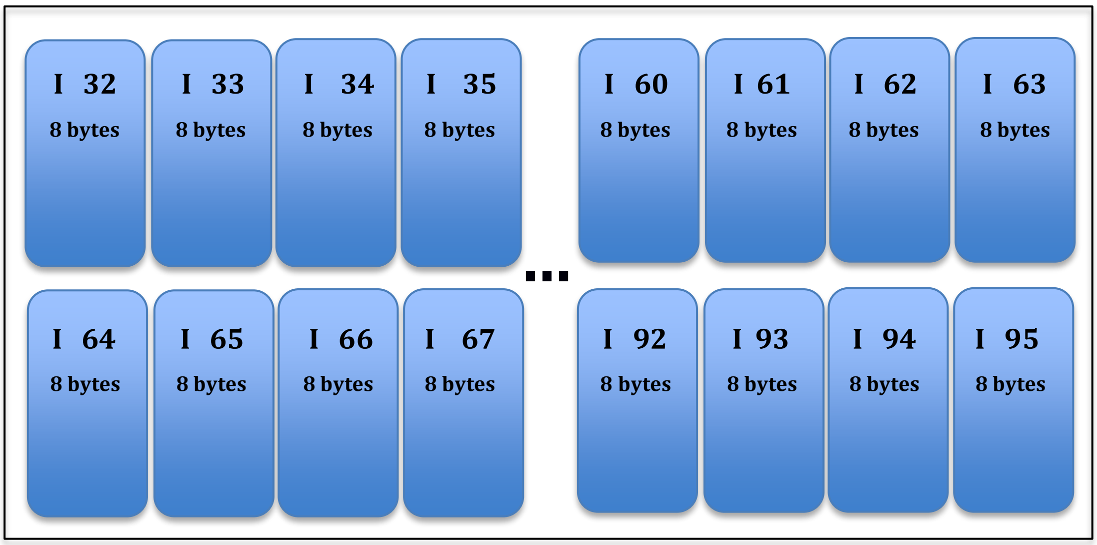

Figure 4: 64 small i-nodes adequate for contiguous files, packed in a block

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 optionally 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 forget the **`mount()`** and **`umount()`** operations.

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

3.2 A note about tstBFS commands and about *specfiles*

We tried to implement a set of commands (see pages 8 and 9 further down in this document) that allows you to do unit tests, i.e., test each operation individually. For example, the **`e`** command that creates a dentry has two parameters: the **`name`** of the file (or subdirectory) tocreate, and its inode number, **`in`**. The command only affects the directory, does not allocate an inode, or set the inode bytemap entry to "used". If you want to do that, you must perform the three operations in the specfile. The only exception for this "rule" (that commands act "individually) are the file creation commands, **`create_C`** and **`create_I`**: they act on the directory, the inode, and the bytemap.

3.3 Bytemaps

Implement the bytemap operations, as suggested by the `ffs_bytemap.{c,h}` files that are included in the ZIP. Test each operation, using the **`tstBFS`** and the appropriate *specfile* (tailor the *specfile* to your tests, may be commenting out the options that you do not have implemented yet. Always use the bytemap print, and check the results). Start by allocating only 1 entry, then proceed to the allocation on N entries.

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

3.4 Inodes

Implement the inode operations, as suggested by the `ffs_inode.{c,h}` files that are included in the ZIP. Test each operation, using the **`tstBFS`** and the appropriate *specfile* (tailor the *specfile* to your tests, may be commenting out the options that you do not have implemented yet. Always use the inode print, and check the results). Start by writing two important functions: `largeInode()`, which given an inode number, decides if it's a large or small inode; and `inode_location()`, which, given an inode number, computes which disk block has that inode, and what is the offset in the block where the inode is located.

Then, proceed to the functions that write and read inodes, and test them!

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

3.5 (OPTION A) ROOT Directory and Files

3.5.1 Directory operations

Implement the directory operations, as suggested by the `bfs_dir.{c,h}` files that are included in the ZIP. Test each operation, using the **`tstBFS`** and the appropriate *specfile* (tailor the *specfile* to your tests, may be commenting out the options that you do not have implemented yet. Always use the directory print command, and check the results).

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

3.5.2 File operations

Implement the two file operations, **`create_C`** and **`create_I`**, as suggested by the `bfs_file.{c,h}` files that are included in the ZIP. 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 file_operations {
…
  int (*create_C)(char *filename, unsigned int maxBlocks);
  int (*create_I)(char *filename);
…
};
```

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

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

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

c) store in the i-node **`ino`** the address of the first and last entries returned on (c) above;

d) call our good old **`dir_ops.create("FC”, in)`**. The END.

3.5.3 Other file operations

We will NOT be implementing other file operations, such as delete, open, close, etc..:

3.6 (OPTION A+) Subdirectory and Files

If you reached this point, congratulations!

Now implement the subdirectory operations, enhancing the `bfs_dir.{c,h}` files that were used to implement the root directory operations. The specfile for subdirectory tests will work as follows (note: **`cwd`** stands for current working directory):

```
O /    Opens the root dir, loading its block to memory; the cwd is now /
#      Start with some dentry creations in the root directory
e AAA 0
i FIND
c SUBD 1 D  Create the subdirectory SUBD in the root
Y           List the files (entries) in the root, SUBD will show with
            an inode >= 32 (remember it’s a small inode)
O SUBD      Opens the subdir SUBD, loading its block to memory; the cwd is
            now the subdirectory SUBD
#Start with some dentry creations in the subdirectory
e AAA 0     Create the dentry, not a problem because in SUBDIR the name
            AAA does not exist (and the inode number 0 is faked, anyway)
Y           List the files (entries) in the SUBD, will show only AAA
            An inode >= 32 (remember it’s a small inode)
C SUBD      Close SUBD, writing its contents to the correct block on disk;
            the cwd is now back to /
Y           List the files (entries) in the root, SUBD will show!
C /         Close the root, writing its contents to the root disk block
```

Congrats!!!

Set of commands used in Mini-Projecto, MANDATORY Part

Command	Description
`C /`	**C**lose the root directory and write the in-memory structure back to disk.
`D disk0 100`	create**D**isk: Create a file named "disk0" with 100 blocks (x 512 bytes). Blocks are zeroed, and the disk device is automatically **closed**.
`F disk0 in`	**F**ormat: Format the "disk0" with an i-node table of **`in`** (must be even) blocks. The superblock (SB) is written with the appropriate values in its entries, and the disk device is automatically **closed**.
`M disk0`	**M**ount "disk0". The disk is **opened**, the SB is read in to memory and the SB contents are printed out, the **mounted** flag is set and the SB written back to disk. The bytemap in-memory data structures are initialized.
`O /`	**O**pen and read the root directory block to in-memory structure.
`U`	**U**nmount the currently mounted disk; the **mounted** flag is cleared and the SB written back to disk, that will be **closed**.
`a in type`	**a**llocate i-node (only) with number **`in`** and type **`type`** (mark it valid and set its type). DO NOT touch bytemap entries.
`d in`	**d**eallocate i-node (only) number **`in`** (clear everything). DO NOT touch bytemap entries.
`e name in`	Create a d**e**ntry in the current directory, store its **`name`** and inode number **`in`**.
`g map qty`	Bytemap **g**et the start location of the 1st free entry on **`map`** where at least the next **`qty`** successive entries are also free. The maps are: 0-small inodes; 1-large inodes; 2-data
`i name`	Create an **i**ndexed file named **`name`**. All metadata structures are updated.
`t name`	Delete a den**t**ry (only) in the current directory. DO NOT touch bytemap entries.
`s map st qty val`	Bytemap **s**et: the **`st`**art and the next **`qty`** successive entries on **`map`** are set to the value **`val`**. The maps are: 0-small inodes; 1-large inodes; 2-data
`B map`	Print the **B**ytemap Table. The maps are: 0-small inodes; 1-large inodes; 2-data
`I val`	Print the **I**-node Table; print the i-node (absolute) number and, if **`val`** is 1, the valid, type and size fields; else print only the i-node (absolute) number and the valid field
`Y`	Print the **ROOT** director**Y** "valid" entries (name and i-node number)

Set of commands used in Mini-Projecto, OPTIONAL Part

(EXTRA or updated commands)

Command	Description
`c name nblocks T`	Create a **c**ontiguous file named **`name`**, with **`nblocks`** reserved (allocated), and type **`T`**. All metadata structures are updated.
`C name`	**C**lose the current directory and write the in-memory structure back to disk. Valid names are / (root directory) and any valid filename (which must be a sub-directory in /). It's the user responsibility to assert that **`name`** corresponds to the **current** dir.
`O name`	**O**pen and read the directory block to in-memory structure. Valid names are / (root directory) and any valid filename (which must be a sub-directory in /). After a successful read, **`name`** becomes the current directory.
`Y`	Print the current directorY "valid" entries (name and i-node number)

1. A short and quick guide to the deliverables & specfile and their outputs

Delivery:

- You must deliver all the `.h` and `.c` files, **<u>except</u>** `disk_driver.{c, h}` and `testBFS.c`, 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.
- You must 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}`
    - OPTIONAL: file operations on `bfs_file.{c,h}`

Specfiles:

- *Specfiles* are delivered, along with their outputs, in the ZIP, located in the directory named **`Specfiles`**; the input specfiles have the extension `.in`, and the corresponding output is in a file with the same name and extension `.out`.