The BFS File System

Introduction to Operating Systems
21.12.2020

Clara Sousa - 58403
cjo.sousa@campus.fct.unl.pt

Tiago Guerra - 58201
tms.guerra@campus.fct.unl.pt

I. Introduction

Apart from developing both the mandatory component of this mini-project and the option A, we have attempted to produce option B; that is, we have developed a main program, **fsckBFS.c** with the aim to improve the cohesion the mini-project, making the several consistency verifications.

II. Managed Inconsistencies

We hereby present the inconsistency situations which have been appropriately taken care of.

1. *mounted* Parameter

There is the need to check if the *mounted* parameter in the superblock is zero, given that this is the only way one can be certain that the disk can be adequately accessed, having no other information previously stored with other files;

2. *fsmagic* Parameter

Ensuring the program may run, verifying if the value of *fsmagic* in the superblock is the same as the one defined by the FS_MAGIC constant;

3. *size* Parameter

Making sure the value returned by the command **stat**, which represents the size of the disk (in number of blocks), is the same as the value corresponding to this same size, saved in the superblock;

4. Sum of Areas in the Disk

It is essential to verify if the sum of the sizes of the different areas of the disk corresponds to the size of the disk (presented in number of blocks). Hence, we have created a variable *sumAreas* which results from the addition of the size of the superblock (1), the size of the directory (1), the number of "user-available" data blocks (*ndatablock*), the number of blocks to store i-nodes (*ninodeblocks*), the number of blocks reserved to store the bytemap for the i-nodes (*nbmapblocksinodes*) and the number of blocks reserved to store the bytemap for data blocks (*nbmapblocksdata*);

5. *isvalid* Parameter

It is essential to check if there are any contradictions between the entries in the bytemap and the corresponding i-nodes regarding this parameter, which reveals if such i-node contains information or not. In order to efficiently implement such verification, we started off by reading the blocks of the bytemap stored for i-nodes and, for each i-node, reading it so as to check if its *isvalid* is different from the corresponding entry in the bytemap;

6. Information Contradictions

In order to check if there were any data entries in the bytemap which contradicted the information in the corresponding i-nodes, we first read the data bytemap and proceeded to go through all the blocks, saving the current index in *index*, and reading each i-node, saving its size in *size*; from there, and only if such i-node is to be valid, do we look through the pointers of the i-node. It is at this stage when a contradiction can be found: if it is either the case that the size of the i-node is smaller than zero and the correspondent data block is equal to 1, or the size of the i-node is larger than zero and the data block presents a value of 0. If no such contradiction is found, the *size* variable is updated by decreasing it with the size of a block. Once all the pointers of an i-node are verified, we move on to a new data block, incrementing *index*.

In an attempt to improve this verification, we have also checked if the positions in the pointers array, *direct*, are correctly initialized for each i-node. In order to do so, we go through every i-node using the same method as described above, and checking if a certain pointer is different from zero when the new and updated *size* variable is smaller or equal to 0 (in which case, an error is printed).