Computer Operating Systems

BLG 312E

Homework 3 Report

Eren Özer
ozerer24@itu.edu.tr

Faculty of Computer and Informatics
Department of Computer Engineering
Date of submission: 04.06.2025

1. Implementation

1.1. Details

As instructed in the project specification, the MiniFS file system was implemented using standard file I/O operations in C to interact with a disk image. The disk image is designed as a 1MB storage area divided into 1024 blocks of 1024 bytes. All metadata and data manipulation is performed with these fixed blocks. The system can be initialized/formatted using the mkfs function. This function clears the disk image, writes a superblock containing metadata (block counts, inode layout), zeroes out the inode table, clears the bitmap that tracks free data blocks, and sets up the root directory inode. The root directory's block is initialized with empty entries, and directory entries for "." and ".." are included for consistency with typical file system behavior.
Each file and directory is represented by an inode. These are fixed in number and stored in predefined blocks. Allocation of inodes and data blocks is done through iterations of the inode array and bitmap block. The bitmap is used to track data blocks, inode usage is tracked using an is_valid field in each inode structure. Various helper functions such as allocInode, freeInode, allocDataBlock, and freeDataBlock are used throughout the system to manage these resources.
Directory operations are implemented through arrays of "DirectoryEntry"s stored in data blocks. Each directory can hold a fixed number of entries per block, and up to four blocks per directory. When creating a directory, the new directory's data block is initialized and linked to its parent via an entry. Similarly, files are created by allocating an inode and updating the parent directory with the mapping from name to inode. There are checks to prevent creating files or directories with the same name.
For handling the paths given for file operations, resolvePath function is used. It performs iterative traversal, starting from the root and resolving each component using directory entries. Using "." and ".." while using the commands is not supported, only absolute paths can be used, however the entries for them exist in directories for structural correctness.
Each file system operation has been implemented as a general function which uses the helper functions for modifying the disk image. Function declarations are exactly the same as in the instructions, return values of functions vary but most of them (if they return a value) return a negative value upon facing an error. Positive values may have different values. For example ls_fs returns the count of the entries in the given directory.
Main function contains a demonstration sequence that executes the main file system operations as stated in the document. Additionally, by using the executable (mini_fs), command line interface instructions can also be used to use file system functions and modify the same disk similarly. Checking the validity of the output for the demo

sequence can be done using the appropriate check target. This is explained in the README file of the project.

1.2. Design Choices

Each directory is initialized with . and .. entries to be more realistic even though relative path parsing is not supported. Only absolute path parsing is possible while performing file operations. Directories and files both share the same inode structure, recognized by the is_directory flag. This allows for simpler allocation logic. A maximum of four direct data blocks per file is used, indirect block handling is not implemented, which keeps block allocation logic simple and easier to handle. Success messages are mostly printed from the main function based on the return value of the function, whereas the error messages are printed into stderr by the file operation functions themselves to clearly indicate what is wrong. Helper functions like allocInode, readInode, writeInode, and block input output utilities were used to avoid code repetition for higher level operations. Main function has a demo sequence combined with a command line interface handling capability, similar to the project instructions.

1.3. Conclusion

During the implementation of the project, I tried to handle different cases and scenerios and use code efficiently, this helped me understand how real file systems are structured internally, even though this was a simple implementation. Also, the fact that most of the header file content was provided in the project document was very appreciated by me as it would have been more confusing without certain structures. Overall, hands on implementation of various operating systems concepts was helpful for me.

2. Questions

2.1. How are directory entries stored/searched?

They are stored as arrays of "DirectoryEntry"s within the data blocks pointed to by a directory’s inode. When creating or searching for an entry, the system iterates through these blocks and checks each entry in the array for a matching name or available slot depending on the operation. Entries with an inode_number of -1 are considered unused. This linear search approach is enough for small directory sizes like the one we are using for this project. The resolvePath function uses this logic internally to traverse each directory level and find the corresponding inode for a given absolute path.

2.2. What happens when the disk or inode table is full?

When the disk or inode table is full, the helper functions (allocDataBlock for data blocks and allocInode for inodes) return -1 to indicate failure. If the disk is full meaning there are no free data blocks: allocDataBlock() scans the bitmap but finds no unused bits, returns -1 to indicate this issue. The calling higher level file system operating function then acknowledges the error like "No free data blocks available." and aborts the operation.
If the inode table is full, allocInode() performs a linear scan of the inode table, but all inodes are marked as is_valid = 1, so it returns -1 to indicate cannot find an empty one. Again, the higher level function outputs the error and rolls back any partially completed work like freeing a data block if it had already been allocated. All functions try to communicate with each other as best as possible to ensure a smooth error handling process.

2.3. How is double allocation prevented?

Double allocation is prevented by checking and updating metadata consistently. Inodes use an is_valid flag, and data blocks are tracked using a bitmap. When allocating either one of them, the functions scan for free entries, marks them as used, and writes the change to disk. Before creating a file or directory, the path is resolved to ensure it doesn’t already exist. These steps ensure each resource is assigned only once until it’s freed.

2.4. One encountered error and solution

One issue I encountered was that after creating the root directory and adding files or directories, running ls_fs("/") didn’t show those. The creation functions were returning success, but the root was empty. After debugging, I realized that while the root inode was initialized correctly in mkfs, its associated data block was not written with empty directory entries. So, ls_fs would read uninitialized memory when trying to list contents.

To fix this, I modified mkfs to also write a fully initialized DirectoryEntry array (with all inode_number fields set to -1) to the root's data block. This ensured that future directory operations like had a consistent starting point to use. After this change, newly created entries under root appeared correctly.