OS Exercise 3: Building a Simple Filesystem

Operating Systems Reichman University

09.06.2025

Submission Notes:

- **Submission Deadline:** Via Moodle by `07.07.2025`. **No late submissions will be accepted!**
- **Collaboration:** This assignment is to be done in pairs (i.e., two students per team).
- **Submission Format:** Submit a single ZIP file containing all the requested files.
- **README:** Include a `README.txt` file at the root of the ZIP. This file must contain the submitting students' information. Each line should follow this format:

  ```
  First name, Last name, Student ID
  ```
- **Working Environment:** This assignment requires the use of an Ubuntu OS with x86 64 bit architecture, be sure to work with the appropriate virtual machine/container/device.
- **Compilation Environment:** The checker uses `gcc13` on Ubuntu 24.04 LTS with the C17 standard (Default in GCC13).
- **Compilation and Execution:** Ensure your code compiles without errors and runs as expected.
  - The course staff will only support compilation via a docker container that uses the provided docker reference file.
  - You are provided with a reference compile script called "build.sh", you may modify it as needed but the tester is going to use it ot compile your code, therefore all commands should be relative.
  - You are provided with an example main file (main.c) that you can use to a usage flow with regard to the FS that you are implementing, it is included in the reference build script.
  - You are provided with a reference "run.sh" script, that runs your compiled code, the tester is going to use it to run your code (with a diffrent main).
  - You must ensure that after executing "build.sh" and then "run.sh" that the program runs correctly.
  - **You must submit build.sh and run.sh**.
- **Testing:** Always Write your own unit tests to validate your implementation.
  - you can also use the given test file: `testfilesystem.c` that contains a `main` function to test basic functionality. **Do not submit** `testfilesystem.c`.
  - You are given a reference script that compiles and runs testfilesystem.c called "compile_and_test.sh". **Do not submit compile_and_test.sh**.
  - If you do not pass all the provided tests your submission will get a failing grade.
- **Covered Material:** This assignment covers topics from File system Presentation Recitation.
- **Header File Modification - You can't modify the header file (including the structs)**

- **Dynamic Memory Allocation:** - Dynamic memory allocation (i.e malloc) is not allowed (and is not needed).
- **Simplifying Assumptions:** You may assume that only single threaded process will use your library, therefore, **thread-safety (mutual exclusion) considerations can be Omitted**.

Introduction

In this project, you will implement a block-based filesystem in C, contained entirely within a single disk-image file. Your implementation, called "OnlyFiles", will be a simplified filesystem that supports file operations without directories. By building your own miniature filesystem—modeled conceptually on something like ext4—you'll gain hands-on experience with the core concepts of: low-level I/O, on-disk data structures, metadata management, and robust error handling.

The filesystem will store its data in a regular file (10MB in size), which acts as a "virtual disk". You will implement a C API that allows formatting the filesystem, mounting and unmounting it, creating and deleting files, and reading from and writing to files.

Learning Objectives

By completing this assignment, you will:

- Understand how filesystems manage data at the block level
- Implement core filesystem operations (create, read, write, delete)
- Work with system-level I/O calls instead of buffered I/O
- Gain practical experience with filesystem metadata structures
- Practice error handling for filesystem operations

Disk Layout

Your virtual disk (10MB) is divided into fixed-size blocks of 4KB each, resulting in a total of 2560 blocks. These blocks are organized into four main regions:

Region	Starting Block	Offset (bytes)	Size	Description
Superblock	0	0	4 KB (1 block)	Global filesystem metadata
Block Bitmap	1	4 KB	4 KB (1 block)	Tracks free/used blocks
Inode Table	2	8 KB	32 KB (8 blocks)	File metadata (256 inodes)
Data Blocks	10	40 KB	9.96 MB (2550 blocks)	File contents

Table 1: On-disk region layout (constants defined in `fs.h`).

Filesystem Components

Superblock

The superblock is the first block in the filesystem and contains critical metadata about the entire filesystem structure. It serves as the "table of contents" for your filesystem.

```c
typedef struct {
    int total_blocks;    // Total number of blocks (2560 for 10MB)
    int block_size;      // Size of each block (4096 bytes)
    int free_blocks;     // Number of available blocks
    int total_inodes;    // Total number of inodes (256)
    int free_inodes;     // Number of available inodes
} superblock;
```

Listing 1: Superblock Structure

When implementing the superblock:

- During `fs_format()`, initialize all fields to their appropriate values
- During `fs_mount()`, read the superblock to verify this is a valid filesystem
- Update `free_blocks` and `free_inodes` when allocating or freeing resources
- Always write the updated superblock back to disk after changing its values

Block Bitmap

The block bitmap keeps track of which blocks are free (0) or in use (1). Each bit in the bitmap corresponds to one block in the filesystem.

```c
// Bitmap is stored as: unsigned char bitmap[MAX_BLOCKS / 8];

// To mark block N as used
bitmap[N/8] |= (1 << (N%8));

// To mark block N as free
bitmap[N/8] &= ~(1 << (N%8));

// To check if block N is used
if (bitmap[N/8] & (1 << (N%8))) {
    // Block is in use
}
```

Listing 2: Bitmap Operations

The bitmap helps you quickly:

- Find free blocks when allocating space for files
- Mark blocks as free when deleting files
- Determine if there's enough space available for a write operation

Inode Table

The inode table contains metadata for each file in the filesystem. Each inode is 128 bytes in size, and your filesystem supports up to 256 inodes (files).

```c
typedef struct {
    int used;                          // 1 if inode is active, 0 if free
    char name[MAX_FILENAME];           // File name (up to 28 chars)
    int size;                          // File size in bytes
    int blocks[MAX_DIRECT_BLOCKS];     // Block pointers (12 max)
} inode;
```

Listing 3: Inode Structure

Key operations with inodes:

- When creating a file, find a free inode and initialize it
- When writing to a file, update the size and block pointers
- When reading from a file, use block pointers to locate data
- When deleting a file, mark the inode as free and release its blocks

Note: With 12 direct block pointers and 4KB blocks, the maximum file size is 48KB.

Data Blocks

The data blocks region stores the actual contents of files. Each file can use up to 12 blocks (as specified by MAX_DIRECT_BLOCKS), which means the maximum file size is 48KB.

API Specification

Filesystem Operations

`int fs_format(const char* disk_path)`

- **Description:** Creates and initializes a virtual disk file and prepares the filesystem.
- **Parameters:** `disk_path` - path where the virtual disk file will be created
- **Returns:**
  - 0 on success
  - -1 on failure (e.g., cannot create file)
- **Actions:**
  - If file exists, it will be overwritten
  - Creates a 10MB file
  - Initializes superblock, bitmap, and inode table
  - Sets all blocks as free except those used for metadata

`int fs_mount(const char* disk_path)`

- **Description:** Loads an existing filesystem from a virtual disk file.
- **Parameters:** `disk_path` - path to the virtual disk file
- **Returns:**
  - 0 on success
  - -1 on failure (e.g., file doesn't exist or invalid filesystem)
- **Actions:**
  - Opens the virtual disk file
  - Reads and validates the superblock
  - Loads necessary metadata into memory

`void fs_unmount()`

- **Description:** Ensures all pending changes are written to disk and closes the filesystem.
- **Returns:** None
- **Actions:**
  - Flushes any cached data to disk
  - Closes the virtual disk file

File Operations

`int fs_create(const char* filename)`

- **Description:** Creates a new empty file in the filesystem.
- **Parameters:** `filename` - null-terminated string (max 28 chars excluding null)
- **Returns:**
  - 0 on success
  - -1 if file already exists
  - -2 if no free inodes available
  - -3 for other errors
- **Actions:**
  - Checks if filename already exists
  - Finds a free inode
  - Initializes the inode with the filename and zero size
  - Updates the superblock (decrease free_inodes)

`int fs_delete(const char* filename)`

- **Description:** Removes a file and frees its blocks.
- **Parameters:** `filename` - null-terminated string
- **Returns:**
  - 0 on success
  - -1 if file doesn't exist
  - -2 for other errors
- **Actions:**
  - Finds the file's inode
  - Marks all of the file's blocks as free in the bitmap
  - Marks the inode as free
  - Updates the superblock (increase free_blocks and free_inodes)

`int fs_list(char filenames[][MAX_FILENAME], int max_files)`

- **Description:** Lists files in the filesystem.
- **Parameters:**
  - `filenames` - pre-allocated 2D array to receive filenames
  - `max_files` - maximum number of filenames to retrieve
- **Returns:** Number of files found (0 to max_files), or -1 on error
- **Actions:**
  - Scans the inode table for used inodes
  - Copies filenames to the provided array
  - Returns the number of files found

`int fs_write(const char* filename, const void* data, int size)`

- **Description:** Writes data to a file, overwriting any existing content.
- **Parameters:**
  - `filename` - target file
  - `data` - pointer to data buffer
  - `size` - number of bytes to write
- **Returns:**
  - 0 on success
  - -1 if file doesn't exist
  - -2 if out of space (not enough free blocks)
  - -3 for other errors
- **Actions:**
  - Finds the file's inode
  - Calculates how many blocks are needed
  - Frees any previously allocated blocks
  - Allocates new blocks as needed
  - Writes data to the allocated blocks
  - Updates the inode (size and block pointers)
  - Updates the bitmap and superblock

`int fs_read(const char* filename, void* buffer, int size)`

- **Description:** Reads file content into a buffer.
- **Parameters:**
  - `filename` - source file
  - `buffer` - pre-allocated buffer to receive data
  - `size` - buffer size in bytes
- **Returns:**

  - Number of bytes read on success
  - -1 if file doesn't exist
  - -3 for other errors

- **Actions:**
  - Finds the file's inode
  - Determines how many bytes to read (minimum of file size and buffer size)
  - Reads data from the file's blocks into the buffer
  - Returns the number of bytes read

Implementation Requirements

System Call Usage

Your code must use low-level system calls to access the virtual disk:

```c
// Opening the virtual disk
int disk_fd = open(disk_path, O_RDWR | O_CREAT, 0644);

// Reading a block
lseek(disk_fd, block_num * BLOCK_SIZE, SEEK_SET);
read(disk_fd, buffer, BLOCK_SIZE);

// Writing a block
lseek(disk_fd, block_num * BLOCK_SIZE, SEEK_SET);
write(disk_fd, buffer, BLOCK_SIZE);

// Closing the disk
close(disk_fd);
```

Listing 4: Required System Calls

Do **NOT** use:

- High-level stdio functions (`fopen`, `fread`, `fwrite`, etc.)
- Memory mapping (`mmap`)
- Dynamic memory allocation (`malloc`, `calloc`, etc.)

Block Access Pattern

For all file operations, you should follow this general pattern:

1. Find the target file's inode (if operation requires an existing file)
2. Calculate which blocks are involved in the operation
3. Read or write those blocks using `lseek` to position and `read`/`write` for I/O
4. Update metadata (inodes, bitmap, superblock) as needed
5. Write changes back to disk

Implementation Steps

We recommend implementing the functions in this order:

1. `fs_format` - Start by creating the disk and initializing structures
2. `fs_mount` and `fs_unmount` - Basic filesystem access
3. `fs_create` and `fs_list` - Simple file management
4. `fs_write` and `fs_read` - Data access
5. `fs_delete` - Resource cleanup

Testing Your Implementation

The provided `main.c` includes a basic test that:

- Formats a virtual disk
- Mounts the filesystem
- Creates a file
- Writes data to the file
- Reads the data back
- Unmounts the filesystem

You should extend this with additional tests:

- Creating multiple files
- Writing files of different sizes
- Filling the filesystem to capacity
- Deleting files and reusing the space
- Testing error conditions (file not found, disk full)

Sample Helper Functions

These are some useful helper functions to consider implementing:

```
// Find an inode by filename
int find_inode(const char* filename);

// Find a free inode
int find_free_inode();

// Find a free block
int find_free_block();

// Mark a block as used
void mark_block_used(int block_num);

// Mark a block as free
void mark_block_free(int block_num);

// Read an inode from disk
```

```c
void read_inode(int inode_num, inode* target);

// Write an inode to disk
void write_inode(int inode_num, const inode* source);
```

Listing 5: Useful Helper Functions

Common Pitfalls

- Forgetting to update the superblock after allocating/freeing resources
- Not writing changes back to disk (especially metadata)
- Buffer overflows with filenames
- Not handling partial reads/writes correctly
- Miscalculating block offsets
- Forgetting to update the bitmap when allocating/freeing blocks

Final Notes

- **Read the Instructions Carefully:** Ensure that you adhere to all restrictions and requirements.
- **Testing:** Test your code thoroughly with various inputs to ensure correctness.
- **Code Quality:** Write clean, well-commented code to help the graders understand your implementation.
- **Error Handling:** Pay special attention to error cases and provide appropriate error codes.
- **Grading Reminder:** A happy grader is a merciful grader!

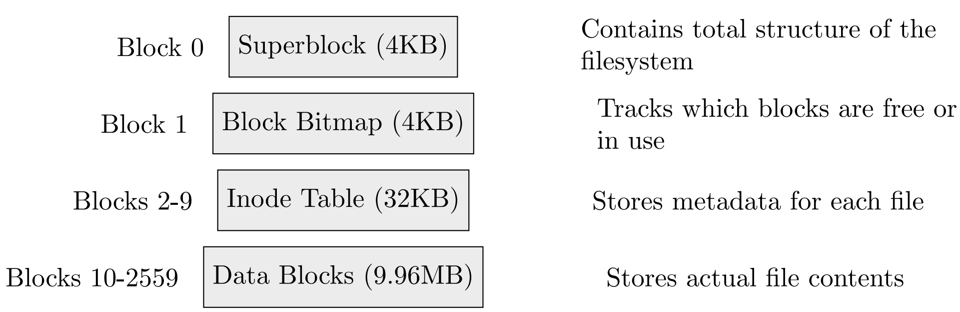

Figure 1: OnlyFiles Filesystem Structure

Good Luck!