Assignment 4 - File System

Operating Systems - Fall '23

Contents

1 Introduction

In this assignment, your task is to create a simplified file system, similar in structure to the modern Linux filesystem, ext4. The goal is to familiarize you with the essential file system interfaces provided by the operating system and their interaction with a disk. While the outcome will be a working file system, it's important to note that it won't include advanced features.

2 Objectives

By now you should be quite comfortable with the C programming Language. Regardless of your background, by the end of this assignment, we hope that you will comfortably and confidently be able to do the following:

- Develop a foundational comprehension of the principles governing file systems.
- Explore the hierarchical organization of directories and files within a file system.
- Gain insights into how file system interfaces facilitate data management and retrieval.
- Apply theoretical knowledge to create a simplified file system resembling the ext4 structure.

By achieving these objectives, you will not only complete the assignment successfully but also acquire a practical understanding of file systems and their integration with operating systems.

3 Overview

This assignment comprises two integral components: the Disk Emulator and the File System.

3.1 The Disk Emulator

Note: The Disk Emulator component has already been developed and provided for your use.

As we delve into constructing a file system, interacting directly with physical disks via their interfaces can be complex and risky. To address this challenge and create a secure environment for your file system, we introduce the Disk Emulator.

The Disk Emulator operates with a disk image, essentially a file serving as a virtual representation of a physical disk. It efficiently manages this file, presenting you with two reliable and safe Application Programming Interfaces (APIs): `disk_read ()` and `disk_write ()` (further details below). For the purpose of this assignment, consider the disk image as an actual disk with multiple blocks, where a block represents the smallest unit that can be read from or written to on a disk.

3.1.1 `int disk_read (uint32_t blocknum, void *buf);`

This API enables you to read any block specified by its block number into the buffer you provide. Throughout your file system implementation, utilize this function whenever you need to retrieve information from a specific block on the disk.

3.1.2 `int disk_write (uint32_t blocknum, void *buf);`

Similar to `disk_read ()`, this API facilitates writing the content of the buffer into the block specified by its block number. Incorporate this function in your file system implementation when you need to store data on the disk.

Note: Both `disk_read ()` and `disk_write ()` consistently operate from the start of the block. Consider the significance of this information.

By understanding and effectively utilizing the Disk Emulator, you establish a robust foundation for seamlessly integrating file system operations with virtual disks.

3.2 The File System

Now, let's talk about the part where we organize and handle all the information on the disk. Here's a quick rundown of what you need to do, along with some extra details:

- **Formatting a Disk:** Get the disk ready to work with your file system.
- **Mounting an Existing Disk:** Make sure your file system can talk to a disk that's already set up.
- **File and Directory Management:** Creating: Make new files and folders. Deleting: Remove files and folders you don't need.
- **Reading/Writing in Files and Folders:** Reading: Grab information from files and folders. Writing: Put new information into files and folders.
- **Listing Contents of Directories:** Find a way to see what's in a folder.

Remember, these are like the superhero tasks of your file system! Each one has its own special powers, but together they make sure all your data on the disk is safe and sound. Don't forget, you've got the super tools `disk_read ()` and `disk_write ()` to help your file system and the disk understand each other better.

You're doing fantastic, and we're here to cheer you on every step of the way!

4 Design and Layout

Before we embark on the next steps, let's take a moment to understand the blueprint of our filesystem. Please refer to the image below for a visual representation:

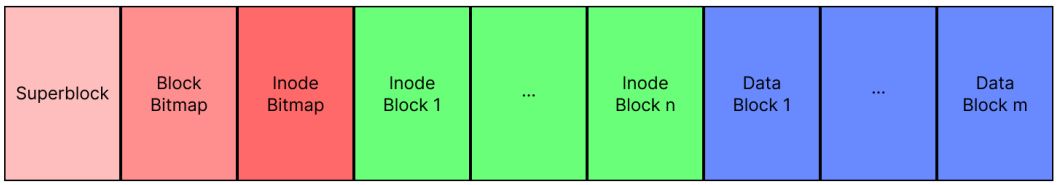

Figure 1: Representation of the file system

Overview

Our filesystem's architecture follows a specific structure:

1. **Superblock (Block 1):** The superhero of our filesystem. It holds crucial information about the whole setup.
2. **Block Bitmap (Block 2):** This is like a traffic light system, telling us which blocks are free and which are in use.
3. **Inode Bitmap (Block 3):** Similar to the block bitmap but for our superhero squad of inodes.

4. **Inode Blocks (Blocks 4 to $n+3$):** Home to our superhero squad, the inodes. Each inode is like a guardian angel for a file or a directory.
5. **Data Blocks (Blocks $n+4$ to $n+m+3$):** Where the actual data of your files and directories lives.

Understanding this layout is like having a treasure map for your filesystem. The Superblock leads the way, the Bitmaps keep things organized, the Inodes guard your files, and the Data Blocks hold the treasures.

Ready for the adventure? Let's explore each part step by step!

4.1 The Superblock:

The Superblock is like the chief architect of our filesystem, containing vital details in a neat little package:

```
struct superblock
{
    uint32_t s_blocks_count;
    uint32_t s_inodes_count;
    uint32_t s_block_bitmap;
    uint32_t s_inode_bitmap;
    uint32_t s_inode_table_block_start;
    uint32_t s_data_blocks_start;
};
```

This compact 24-byte superhero holds the following information:

1. **Total Blocks:** `s_blocks_count`
   - The number of blocks in the entire file system.
2. **Total Inodes:** `s_inodes_count`
   - The count of inodes, each acting as a guardian for a file or directory.
3. **Block Bitmap Pointer:** `s_blocks_bitmap`
   - The pointer to the block bitmap, indicating which blocks are free or in use.
4. **Inode Bitmap Pointer:** `s_inode_bitmap`
   - The pointer to the inode bitmap, keeping track of used and free inodes.
5. **Inode Table Start Pointer:** `s_inode_table_block_start`
   - Points to the start of the inode blocks, where each inode resides.
6. **Data Blocks Start Pointer:** `s_data_blocks_start`
   - Marks the beginning of the data blocks, where the real treasures (data) are stored.

Now, let's explore the supporting characters in our filesystem saga:

4.2 The Block Bitmap:

It's as simple as it gets—one integer per block, waving either a 0 for free or a 1 for in-use. Like a traffic light system, it guides us through the blocks, letting us know where we can build and where we can't.

But what are those blocks? The following `union` represents a single block. `unions` are different from your usual `structs`, and can only hold one of the fields at a time, instead of storing all the fields at the same time (like `structs` do.)

```
union block
{
    struct superblock superblock;                          // Superblock
    struct inode inodes[INODES_PER_BLOCK];                 // Inode block
    uint32_t bitmap[FLAGS_PER_BLOCK];       // Bitmap block (inode or data)
    struct directory_block directory_block;                // Directory block
    uint8_t data[BLOCK_SIZE];                              // Data block
    uint32_t pointers[INODE_INDIRECT_POINTERS_PER_BLOCK]; // Indirect pointer block
};
```

`sizeof(union block)` will be the size of the largest field in the `union`. You should try to keep the size of all fields consistent, so all block types are of the same size.

A generic block can contain one of the following:

1. Superblock
2. Collection of `inodes`
3. Bitmap
4. Directory
5. Data
6. Indirect pointers

The fields of the `union` reflect this. Here's a brief description of them.

1. **Superblock**: `superblock`
   - If a block is a superblock, only the superblock field will be valid. It holds essential information about the whole filesystem.
2. **Inodes array**: `inodes[INODES_PER_BLOCK]`
   - If a block is an inodes block, the inodes array field becomes active. It's a collection of inodes for files and directories. (Total number of inodes that can be fit inside the block depends upon the size of the inode, and the size of the block.)
3. **Bitmap**: `bitmap[FLAGS_PER_BLOCK]`
   - A block can serve as a "bitmap" block, indicating available data or inode blocks. In this case, only the bitmap field will be valid, containing an array of 4-byte flags. (If you think this is a missed optimization, it can be but we are not limited by space constraints, and that's the only thing it effects.)
4. **Directory Block**: `directory_block`
   - If a block is a directory block, it stores a `struct directory_block` instead of the other fields. It's like a folder containing information about files.
5. **Data block**: `data[BLOCK_SIZE]`
   - A block can simply be an array of raw bytes, storing pure data. This is a data block, like a treasure chest filled with information.
6. **Indirection Pointers**: `pointers[INODE_INDIRECT_POINTERS_PER_BLOCK]`
   - A block can also be an "indirect" block, storing pointers to other data blocks instead of raw data bytes. The pointers field becomes active in this case. Note that we are using 4-byte pointers, to refer to other blocks. Think about how this limits the maximum file size available to us.

4.3 The Inode Bitmap:

Similar to its sibling, the Inode Bitmap follows the same pattern: Again, a straightforward system. Each integer tells us whether a guardian inode is on duty (1) or taking a break (0).

The inode bitmap refers to inodes, but what are those individual indoes. An inode is like the guardian angel for a file or directory, keeping important information in check. Here's a closer look at its components:

```
struct inode
{
    uint64_t i_size;
    uint32_t i_is_directory;
    uint32_t i_direct_pointers[INODE_DIRECT_POINTERS];
    uint32_t i_single_indirect_pointer;
    uint32_t i_double_indirect_pointer;
};
```

Following is a brief description of each of the fields of this `struct`.

1. **Size of the file**: `i_size`
   - This field holds the size of the file stored in the block. It's like knowing how much space the file takes up in the filesystem.
2. **isDirectory Flag**: `i_is_directory`
   - The flag represents whether this block corresponds to a directory or not. A block can be a file block or a directory block.
3. **Direct Pointers**: `i_direct_pointers[INODE_DIRECT_POINTERS]`
   - These are like signposts pointing directly to data blocks. A file can have multiple data blocks, but it's limited to a maximum of `INODE_DIRECT_POINTERS` blocks. Each pointer guides us to a specific block containing data.
4. **Single indirect pointer**: `i_direct_pointers[INODE_DIRECT_POINTERS]`
   - This pointer refers to a special block that contains pointers exclusively pointing to other data blocks. It's like having a list of addresses, each leading to a different block.
5. **Double indirect pointer**: `i_direct_pointers[INODE_DIRECT_POINTERS]`
   - This pointer refers to another special block. Each pointer in this block points to yet another block that contains pointers. It's like a cascade of addresses, leading us from one level to another:

     `Double Indirect Pointer -> Block containing pointers ->`
     `Each pointer to block containing pointers -> Each pointer to a data block.`
   - This allows us to reach multiple data blocks, creating a more extensive network of information.

Understanding the inode is like having the blueprint for a file or directory—knowing its size, type, and the paths to its data blocks.

5 Specification

The file `fs.h` declares an API to interact with the file system directly. You can think of this as a low-level API that interacts directly with your hard disk, but for everyone's sanity, we will be using virtual hard disk images. These are flat binary files, on which you write raw binary data, and treat it as the data on a hard disk.

You are to implement the following functions. Their detail is provided.

5.1 int fs_format()

Formats the ~~mounted~~ disk image with a new file system. This process involves replacing the previous superblock with a new superblock, with the updated information as well as creating the `'root'` directory. The function returns 0 on success, -1 on failure. Format should also initialize bitmaps, as well as inode blocks.

5.2 int fs_mount();

This function mounts the filesystem. Mounting involves reading the superblock, extracting the useful information regarding the disk out of it, and putting it in the memory of the filesystem program. The function returns 0 on success, -1 on failure.

5.3 int fs_create(char *path, int is_directory);

Create a file or directory at the specified path, represented by `path`. Note that this path can only be an absolute path. An absolute path is the one that exactly defines the path of a file or a directory, starting from the `root` directory. If intermediate directories leading to the path do not exist, these must be created before finally creating the file in the correct directory. The `is_directory` flag is used to differentiate between whether to create a file or a directory on the disk.

5.4 int fs_remove(char *path);

This function removes the specified file or a directory. `path` must be an absolute path, and returns 0 on success, and -1 on failure. If the path represents a directory, remove all files and directories inside it recursively (does NOT mean you are required to use recursion). The provided path must start with a slash (/) and be absolute.

5.5 int fs_read(char *path, void *buf, size_t count, off_t offset);

This function reads from the disc, the specified file, and puts it in a buffer (specified by `buf`). Offset represents the offset at which offset (in bytes) to start reading from, inside the data block of the file. Returns the number of bytes read, in case of success.

5.6 int fs_write(char *path, void *buf, size_t count, off_t offset);

This function writes data from the buffer pointed to by buf to a file at the specified path. It also creates the file if it doesn't exist already.

5.7 int fs_list(char *path);

Lists all the files inside a directory at the specified path. If path is not a directory, it should fail.

5.8 void fs_stat();

A helper function, that displays statistics about the file system. You can use this function to dump the state of the filesystem, or read the superblock etc.

6 Setting up Docker

Although the assignment can be built on any Linux system, we recommend using Docker to build and run the project. The project, as well as the test files, have been extensively tested on a docker image, and we can guarantee that it would work on the docker image irrespective of your environment. Therefore, your submission will be tested on the docker container provided. So, we recommend using Docker to build and

run the project. If you are not familiar with Docker, you can read about it here. You can install Docker on your system by following the instructions here. Once you have installed Docker, you can build the docker image by running the following command in the root directory of the project.

1. Install docker on your machine
2. Remove previous containers by running `docker kill os-fall-2023; docker rm os-fall-2023` in the terminal.
3. Open a terminal and navigate to the root of your handout folder.
4. Build and run the docker container by typing `docker-compose run -rm os-fall-2023` in the terminal (Note: Double dash before rm).
5. The container has the following packages installed: `build-essential, valgrind, binutils, and git`. You can install more packages if you need to (e.g. `apt-get install vim`).
6. To exit the container, type '`exit`' in the terminal.

7 Build System

For building the project, you should use the `Makefile` provided. To build the project, run the following command in the project directory, (containing the `Makefile`):

```
$ make
```

This would create your memory management library, inside the `build` directory, in `debug` mode by default. Debug mode is helpful for debugging, as it provides necessary macros, and useful flags to the compiler, and also disables compiler optimizations. You can also build the project in `release` mode, which enables compiler optimizations, and disables debugging macros. To build the project in `release` mode, run the following command:

```
$ make clean; make BUILD_DEFAULT=release
```

Or, you can change the value of `BUILD_DEFAULT` variable in the `Makefile` to `release`, and then run `make` without any arguments. Always make sure to run `make clean` before switching between the build modes, as the object files are not compatible between the two modes.

To clean the `build`, or to force a rebuild from scratch, use `make clean`, or `make clean; make`.

Please note that the provided `Makefile` assumes the provided directory structure. So, the above instructions are only valid if you stick to that structure.

8 Testing

The filesystem code gets compiled as a static library, which can then be included with other applications, or test programs. This has already been set up for you in the `Makefile`. In order to build the file system, run:

```
$ make test
```

We have also provided a `driver.c` file, which you can use to test the implementation of your file system. Use the following command to run this file.

```
$ make driver
```

There is also a shell utility available, to interact with the file system using a command line interpreter. This program uses the underlying functions, that you'll implement, to do various tasks. Use the following to run the shell program.

```
$ make shell
```

The shell program, however, expects a disk name, as well as the total number of blocks in the disk program. If the specified disk image doesn't exist, the program creates a new image. You can pass these arguments to the shell via the following command

```
$ make shell ARGS="<disk name> <number of blocks>"
```

Run `help` inside this shell to see the list of available commands and their syntax. A brief description is provided as follows:

- `exit`: exit out of the shell
- `help`: lists the available shell commands
- `format`: formats the file system
- `mount`: mounts the file system
- `stat`: check the file system's superblock status, as well as some other debug information that you can add
- `ls <path>`: lists all the files and directories present inside the directory specified by <path>
- `cat <path>`: reads a file specified by <path> and displays its contents
- `delete <path>`: deletes the file/directory specified by <path>. note that this should delete the file from the file system (or the disk image as well)
- `copy_in <local_path> <fs_path>`: copies a file from the local file system to your virtual disk's file system. <fs_path> is an absolute path referring to the file to write in the virtual disk's file system.
- `copy_out <fs_path> <local_path>`: copies a file from the disk's virtual file system to your local file system. <fs_path> is an absolute path in the virtual file system, while <local_path> can be relative or absolute, it is the file name to write to in the local file system.

9 Submission

The submission should be done through LMS on the assignment tab. **fs.c is the only file that you need to submit.** Please don't include any other files, or directories, as they'd just be making our lives harder, and would be removed nonetheless. Rename your file to `<rollnumber>.c`, so for example, `fs.c` would become `24100173.c` if that's your roll number.

Late Submissions Will Not Be Accepted

Good Luck, and Happy Coding!