1 Assignment

Using the descriptions in the sections that follow, implement ExFS2 in C, using any standard libraries. The file system is to reside not in a partition on a disk, but rather as an extensible (the Ex in ExFS2) collection of fixed-size files. You will construct a program corresponding to these specifications with command-line options for writing, reading, removing, and listing directories and files contained in the file system.

2 Format

You’ll use regular linux files as storage for your file system. The contents of your file system will be placed in two types of files:

1. inode segments - storage files containing inodes and directory data; and
2. data segments - storage files containing data blocks of stored files.

Segment size will be fixed to 1MB for each segment.

2.1 Directory Segments

Each directory segment will contain the following information about its portion of the file system namespace: an inode map (denoting which inodes are in use and which are free – a bitmap) and corresponding inodes. The first inode of the first inode segment should correspond to the root directory. When a directory segment is created, ensure that the inode map is initialized appropriately.

When files and directories are added, search the inode segments sequentially until you find the first available inode (inodes will be made free when files and/or directories are removed and should be reused). When no free inodes are found in the existing inode segments, a new inode segment should be initialized.

Each directory and each file should be reachable by providing its path and file name in ExFS2. For example, reconstructing the stored data for file stored in ExFS2 at /dir1/dir2/file.txt should require at least the following actions:

1. Read the first directory segment to fetch the root directory inode.
2. Use the root's inode to fetch the data blocks (from data segments) that constitute the data of the root directory.
3. Search through the root directory's entries for "dir1" to find its inode number.
4. Fetch the inode for "dir1" and fetch the data blocks that constitute the directory data for "dir1".
5. Search through the "dir1" directory's entries for "dir2" to find its inode number.
6. Fetch the inode for "dir2" and fetch the data blocks that constitute the directory data for "dir2".
7. Search through the "dir2" directory's entries for "file.txt" to find its inode number.
8. Fetch the inode for "file.txt", then its data blocks, and output their contents.

Each directory segment should link to the next directory segment (think linked-list, but without pointers – since your file system will run on top of the system's file system, you can use a segment's file name to indicate the next directory segment).

File and directory names should be contained in directory entries only. Directory entries should only be found in data segments.

Directory segment structure:

```
<- 0 byte                                                    1MB  ->
+-------------------------------------------------------------------+
|  Free   | Inode | Inode |                                         |
|  inode  |   x   |  x+1  |  etc...                                 |
|  list   | block | block |                                         |
+-------------------------------------------------------------------+
```

2.2 Data Segments

Data segments will contain a free block list and as many 4096-byte blocks as will fit in the 1MB segment after accounting for the free block list.

When adding a new file, assign the needed blocks by sequentially searching the existing data segments for the first free block. If a free block is not found within the existing data segments, a new data segment is created and its free block list initialized.

The most straightforward free list might be implemented as a bit map of sufficient size to provide a bit for each block in the file system, specifying whether the corresponding block is free or used.

Inodes should initially be unused and initialized as such. The blocks that are used for file data storage will appear after the free list. No action is required for the blocks themselves, since the only way they might be read or written is when the file system structures like the free list or inodes refer to them.

Do NOT turn in your segment files with your solution.

Data segment structure:

```
<- 0 byte                                                    1MB  ->
+-------------------------------------------------------------------+
|  Free   | Data  | Data  |                                         |
|  block  | Block | Block |  etc...                                 |
|  list   |   x   |  x+1  |                                         |
+-------------------------------------------------------------------+
```

3 Inodes

Each inode should fit into a single 4096 byte block. Your inodes are to to contain as many direct block references as will fit after you account for attributes (at least the file size should be included or you won't know when to stop reading the last block), one single indirect block, one double indirect block, and one triple indirect block. You are not reproducing a user environment, so you also don't have to store the user, group and permissions information. I also don't care about time stamps. Just focus on what will allow you to read and write a file in its entirety.

4 Functionality

Your program should allow files to be added to your file system, read, and removed from it, and for the contents of the file system to be listed. It might be easiest to describe the intended functionality by providing examples of each. For example, the following command line invocation should list the contents of your file system starting with the root directory:

```
./exfs2 -l
```

The output might be a file system tree, showing the names of each directory and file contained within your file system (think about recursing through it starting with the root, adding a number of leading tabs to each line of output equivalent to the depth of recursion).

The following invocation would add the specified file to your file system (as well as each directory in the path):

```
./exfs2 -a /a/b/c -f /local/path/to/example
```

In this example, /a/b/c is the path at which you intend the input file to reside in your file system. The intended result is that you will create exfs2 directories a, a/b, and a/b/c, then create a file named example in directory c and copy the contents of the input file at /local/path/to/example into exfs2. Each directory in the path should be created if it does not already exist, resulting in an inode being assigned for each directory that is created, and a directory entry added to the directory's data block (if it's a new directory, you'll also need a new data block for its inode to point to).

The following invocation would remove the specified file from your file system (as well as any directory that becomes empty after you remove the specified file):

```
./exfs2 -r /a/b/c/example
```

In this example, the file a/b/c/example should be removed from your file system if it exists. Leave the directories even if they're empty. If the end of the provided path is a directory and not a regular file, then remove the directory and all files contained within it.

The following invocation would read the contents of the specified file to stdout, then redirect that output to file foobar.

```
./exfs2 -e /a/b/c/example > foobar
```

Reading a file's contents in this manner should NOT result in the file being removed from your file system. File data should be written to stdout. There should be no difference between the original file and the data returned by your file system, regardless of whether that data is text-based or not. If the provided path ends in a directory, do not write its data to stdout, but notify the user that they should specify a regular file and exit.

Think of adding a debugging option that would print each inode in the path and the data they point to. For example, in case of directories, print the name of each file and the associated inode's number, in case of files, just print that it is a regular file.

```
./exfs2 -D /a/b/c/example
```

I An output like the following (unrelated to the above example command) might be useful to you as you debug:

```
directory '/':
 'a' 2
 'd' 5
 'e' 6
directory 'a':
 'b' 3
 'f' 7
directory 'b':
 'c' 4
```

5 Hard Requirements

- Your file system should use files in the host machine's file system as storage for your filesystem. Each segment resides in the host machine's file system as a 1MB file. Segment storage files should be the only files your file system creates and manages. NO POINTS will be awarded if your program simply moves files around the host machine's file system instead of breaking them down into blocks and storing them as described in these specifications.
- Rephrasing the requirement above, because it is imperative: your file system should be self-contained; i.e. metadata and data comprising the file system should be inside of your segment files, formatted according to these specifications.
- Exfs2 should work for any kind of file, text or binary (use fread/fwrite and not string functions).
- A file stored in your file system should, when extracted, match the original file. You can test this by extract the file and perform a diff on the original and extracted file to make sure there are no differences.
- Include an up-to-date Makefile. I don't want to have to reproduce your build.
- Include a README file that contains any information that would be useful for grading: group members, testing, configuration, compiling, etc. (This would be a good place to describe what works, what doesn't, and how the stuff the works does what it does).
- Make sure it compiles and runs before you turn it in.

6 Grading

- 10 points if there is a reasonable attempt at a solution conforming these specifications. (**Any solution NOT conforming to these specifications will receive 0 points regardless of completeness**.)
- 10 points if the segment files are correctly formatted and used.
- 20 points if directories can be added to your file system.
- 10 points if additional directory segments are correctly created and used when available directory segment inodes are exhausted or when the directory segment runs out of free data blocks.
- 10 points if files can be added to any directory in your file system.
- 10 points if files exceeding the size of storage covered by direct block references in inodes result in a singly indirect block being correctly created and used (file is correctly written to your file system and can be correctly extracted).
- 10 points if additional segments are correctly created and used when there are no available directory segment inodes or when there are no available free blocks in data segments.

- 10 points if files/directories can be removed from the file system. (listing contents of the file system no longer shows removed files)
- 10 points if files can be extracted from the file system. (file data returned via stdout matches the file data of the original that was previously added to your file system)
- (Extra credit 10 points) double indirect block pointers are correctly used when an inode's block pointers and indirect block are exhausted.
- (Extra credit 10 points) triple indirect block pointers are correctly used when an inode's block pointers, indirect block, and double indirect blocks are exhausted.

7 What to Turn In

A .tgz of your project directory, excluding segment files.