CS514 Spring '24 - ExFS

1 Assignment

Using the descriptions in the sections that follow, implement ExFS 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 ExFS) 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

Storage for your file system will consist of two types of files: 1. files containing directory inodes and directory data – let's refer to these as directory segments, and 2. files containing (non-directory) file inodes and file data – call these data segments.

Segment size will be fixed to 5MB for each segment.

Each directory segment will contain the following information about its portion of the file system namespace: superblock, free block list, inodes, and data blocks. The superblock, free block list, and inodes should be initialized/formatted each time a new segment is created and updated appropriately as the segment is used for storage. Directory segments should store only directory information and if a directory segment runs out of either inodes or storage space, a new directory segment will be created to accomodate the addition. 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 file name to indicate the next directory segment). There should be one directory segment acting as the entry point to your file system. Inodes in the directory segments may point to regular files (whose own inodes and data will reside in one or more data segments) or to other directories (whose own inodes and data will reside in some directory segment). It would be reasonable that within directory segments, block sizes should be sized to a multiple of the size of each directory entry. Directory entries should contain a name, type (file or directory), and inode number. You may keep names within a fixed-size array.

Similarly, each data segment will contain the following information about its portion of the file system: superblock, free block list, inodes, and data blocks. The superblock, free block list, and inodes should be initialized/formatted each time a new segment is created and updated appropriately as the segment is used for storage. Data segments should not contain directories or their data; that information belongs in the directory segments. Blocks in the data segments should be fixed to 4096 bytes.

The 5MB files that will store your segments will need to be formatted to provide some basic information about your file system each time it is opened by your program. The formatting will result in a superblock, free list, and inodes being written to your segment file. The superblock defines the file system, specifying the number and size of blocks in the segment, the size and number of inodes, and whatever useful information you see fit to include.

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. The blocks that are used for file data storage will appear after the superblock, free list, and inodes. 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.

Segment structure:

```
<- 0 byte                                                     5MB  ->
+-------------------------------------------------------------------+
|              |  Free   |        |                                 |
|  Superblock  |  block  | Inodes | Data ...                        |
|              |  list   |        |                                 |
+-------------------------------------------------------------------+
```

3 Inodes

There should be 10 inodes in each segment and 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, 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:

```
./exfs -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):

```
./exfs -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 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 to your file. 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):

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

In this example, the file a/b/c/example should be deleted 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.

```
./exfs -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. There should be no difference between the original file and the data returned by your file system. The contents of the specified file should be written to stdout (which, in the example invocation, this would cause the contents of /a/b/c/example to be written to stdout then redirected to a local file named foobar). 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.

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

— 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 5MB 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 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 a your segment files, formatted according to these specifications.
- Your file system should work for any kind of file, text or binary (use fread/fwrite and not string functions).
- Your file system should be able to support directory/file names up to 50 characters.
- 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 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.
- 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 file segments are correctly created and used when available file segment inodes are exhausted or when the file segments run out of space.
- 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) 25 points if your file system reuses existing blocks when storing new files with similar contents (think Venti).
- (Extra credit) 25 points if your file system ensures that shared blocks are only removed when the last reference to them is removed.

7 What to Turn In

A .tgz of your project directory.