CS 214 Programming Assignment 3

File Duplicate Detector (100 points)

In this assignment you will write a C program to detect duplicate files in a file system sub-tree.

Recall the Unix filesystem's abstracts a graph of nodes. Nodes can be a regular file, which is data as a byte array, a directory, which is a container of other nodes, devices, or kernel variables.

Duplicate detection seeks to find copies of data in the file system. That is, which files are the same, byte-for-byte? Once such files have been identified, the primary use for duplication detection is freeing up storage space by performing deduplication. A second use of duplication detection is to perform recovery in cases of accidental or malicious deletion.

In this project you will scan the filesystem looking for true duplicates. A true duplicate is one where the blocks on the storage media, that is, an SSD or hard drive, are actual duplications. However, even if two files appear the same, they may not be a true duplicate, but rather share the same underlying blocks in the storage system. Recall that in the Unix filesystem, two files can have the same name in the filesystem graph. Such internal aliases are exposed to the system programmer as the file system's internal data structure of an `inode` is visible at the user level via the `stat` call.

Your task is to report the true duplicates in a subtree of the filesystem. One method of identifying a true duplicate is to use hashing to come up with a single large number that changes when the bytes of a file change. Ideally, even if two files only differ by 1 bit in 1 byte, the hash values should be very different.

In order to standardize the hashIDs for a file, your program must use the MD5 algorithm for hashing. MD5 is a standardized algorithm for providing collision free hashing.

Support Files:
`uthash.h` : A hash table implemented as macros. You can use these, or write your own hash table if you prefer.

`detect_dups.c, detect_dups.h`: The skeleton code and function signatures. You may decide to use other function signatures or code strategies if you prefer, as long as the output is in the correct format.

create_tests.zip: A zip folder containing scripts that create sub-trees to test your code against.

Example Test Results: A folder with example test results. Of course the exact inode numbers will be different on your machine, although the MD5 hash results should be the same.

Output Format:

Your output should be in the following format,

```
File <number>:
   MD5 Hash: <MD5 hash>
   Hard Link (<reference count>): <Inode number>
       Paths:  <Path 1>
               ...
               <Path N>

       Soft Link <number>(<reference count>): <Inode number>
           Paths:  <Path 1>
               ...
               <Path N>
```

`File <number>:` This represents a unique file. For example, if file1.txt and file2.txt contain the exact contents, they will appear under `File n`; however, if they contain different contents, they will appear under `File n` and `File n+1`.

`Hard Link:` This represents a unique inode number. For example, if file1.txt and file2.txt were created by the user with the same contents, they would have different inode numbers. However, if they are hard links pointing to the same inode number, the reference count would increase, and both files would be under the same inode number.
Different Inode Number,

```
Hard Link (1): 38974691
Paths:     file1.txt
Hard Link (1): 28736422
Paths:     file1.txt
```

Same Inode Number - reference count increased,

```
Hard Link (2): 38974691
Paths:     file1.txt
           file2.txt
```

`Soft Link:` This represents a link to a unique “Path.” Since a soft link is unique to each file, it will be under each `Hard Link`. There could be multiple soft links, and each soft link could contain a hard link.

`Paths:` This represents the path in which your hard or soft links exist.

You could use the following template for printf,

```
"File <number>:"
"\tMD5 Hash: <hash>"
"\t\tHard Link (<count>): <inode>
"\t\t\tPaths:\t<Path 1>"
"\t\t\t\t<Path N>"
"\t\t\tSoft Link <number>(<count>): <inode>"
"\t\t\t\tPaths:\t<Path 1>"
"\t\t\t\t\t<Path N>"
```

Error Handling:

You can use `fprintf and stderr` to handle errors.

Standard Error Messages:

If no directory is given, print the following error message, and exit with the text “failure”.

```
Usage: ./detect_dups <directory>
```

If an incorrect directory was given, exit with the text “failure”.

```
Error <error number>: <directory> is not a valid directory
```

Sample Output:

```
File 1
       MD5 Hash: e711f198c6af16d1d13c99f030173add
       Hard Link (2): 409929280
               Paths:  file1.txt
                       file1h.txt
               Soft Link 1(1): 409929290
                       Paths:  file1s.txt
               Soft Link 2(1): 409929374
```

```
                        Paths:  file1hs.txt
        Hard Link (1): 409929281
                Paths:  file2.txt


File 2
        MD5 Hash: 4fe82b0e520f6c2ff023c3e41f6b5c70
        Hard Link (3): 409929098
                Paths:  file3.txt
                        file3h.txt
                        file4.txt
                Soft Link 1(1): 409929270
                        Paths:  file3s.txt
                                file3hs.txt
                Soft Link 2(1): 409929490
                        Paths:  file4s.txt
```

Example Usage:

Executable followed by the directory,

```
./detect_dups <dir>
```

Example,

```
./detect_dups test1
```

Submission Requirements:

You must provide a makefile where the default rule builds an executable called “detect_dups”. The detect_dups program takes a single argument which is a directory to detect duplicates. The program must output to standard output (e.g. printf) in the format as described above.

The program must search the entire sub-directory. However, if a symbolic link points outside the sub-directory it should not be followed.

If you are working in a group, you **must submit as a group in Gradescope**. See the link below for instructions.

Grading:

Your program will be tested against 10 test cases in increasing order of difficulty. Each test case is worth 10 points for a total of 100 points. Two test cases are included in the *example* section of this write-up.

How to add group members in gradescope:
https://help.gradescope.com/article/m5qz2xsnjy-student-add-group-members