Donghyun Park

Prof. Hakner

ECE 357

10/8/2017

PSet 2: Problem 1 – Filesystem Scavenger Hunt

Source Code

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <dirent.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <stdlib.h>

#include <fcntl.h>

//general error handling function

void printError(char \*msg,const char \*file, char \*detail, char \*type){

if(!detail){

fprintf(stderr, "Error: %s\n", msg);

}else{

fprintf(stderr, "%s: %s [%s] - %s \n", type, msg, file, detail);

}

}

//checks the content of file1 and file2 on a buffer size basis

int checkFile(const char\* file1, const char\* file2){

int fd1, fd2, file1Read, file2Read, diff;

char fileBuf1[BUFSIZ];

char fileBuf2[BUFSIZ];

memset(fileBuf1, 0, sizeof(fileBuf1));

memset(fileBuf2, 0, sizeof(fileBuf2));

if((fd1 = open(file1, O\_RDONLY)) == -1){

printError("Failed to open file!", file1, strerror(errno), "Fatal");

return -1;

}else if((fd2 = open(file2, O\_RDONLY)) == -1){

printError("Failed to open file!", file2, strerror(errno), "Fatal");

return -1;

}

while((file1Read = read(fd1, fileBuf1, sizeof(fileBuf1))) != 0 && (file2Read = read(fd2, fileBuf2, sizeof(fileBuf1))) != 0){

if(file1Read == -1)

{

printError("Failed to read file!", file1, strerror(errno), "Warning");

return -1;

}else if(file2Read == -1){

printError("Failed to read file!", file2, strerror(errno), "Warning");

return -1;

}else if((diff = memcmp(fileBuf1, fileBuf2, BUFSIZ)) != 0){

//1 returned means the contents are different

return 1;

}

}

if(close(fd1) == -1){

printError("Failed to close file!", file1, strerror(errno), "Warning");

}else if(close(fd2) == -1){

printError("Failed to close file!", file2, strerror(errno), "Warning");

}

//0 returned means the contents are the same

return 0;

}

//takes in the directory given and recursively goes through the directory to compare files

int parseFile(const char \*dirName, const struct stat targetStat, const char \*fileName, int permissions){

DIR \*directory;

struct dirent \*entry;

struct stat newStat;

char \*symbuf;

ssize\_t link; //0 means not executable, 1 means executable

if(!(directory = opendir(dirName))){

printError("Not a valid directory to open!", dirName, strerror(errno), "Fatal");

return -1;

}

//recursive process

while((entry = readdir(directory)) != NULL){

char path[1024];

sprintf(path, "%s/%s", dirName, entry->d\_name);

//if an item in an opened directory is not stat-able, for some reason, then skip to next item

if(stat(path, &newStat) == -1){

printError("Failed to take stat of following file or directory, skipping", entry->d\_name, strerror(errno), "Warning");

continue;

}

//depending on the type of the directory item returned do different things

switch(entry->d\_type){

case DT\_DIR:

//skipping . and .. for efficiency

if((strcmp(entry->d\_name, ".") == 0) || (strcmp(entry->d\_name, "..") == 0)){

continue;

}

//checking if the current directory being looked at is executable by others. read is not required since the other user has the path to the file

if((newStat.st\_mode & S\_IXOTH) && permissions == 1){

parseFile(path, targetStat, fileName, 1);

}else{

parseFile(path, targetStat, fileName, 0);

}

break;

case DT\_REG:

//in the case of a regular file, peform switch statement depending on a regular file or symlink

switch(newStat.st\_mode & S\_IFMT){

case S\_IFREG:

if(targetStat.st\_dev == newStat.st\_dev && targetStat.st\_ino == newStat.st\_ino){ //checking device number and inode number

printf("%s HARD LINK TO TARGET", path);

//checking read permission by other in hard link cases

if((newStat.st\_mode & S\_IROTH) && permissions){

printf(" OK READ by OTHER\n");

}else{

printf(" NOT READABLE by OTHER\n");

}

}else if(targetStat.st\_size == newStat.st\_size){ //checking byte size of original file to found file

if(checkFile(fileName, path) == 0){

//checking read permissions by other in duplicate cases

printf("%s DUPLICATE OF TARGET (nlink = %lu)", path, newStat.st\_nlink);

if((newStat.st\_mode & S\_IROTH) && permissions){

printf(" OK READ by OTHER\n");

}else{

printf(" NOT READABLE by OTHER\n");

}

}

}

break;

case S\_IFLNK:

//dealing with symlinks, not really necessary, just here for redundancy. Exact same code as case DT\_LNK.

symbuf = malloc(newStat.st\_size + 1);

if((link = readlink(path, symbuf, newStat.st\_size +1)) == -1){

printError("Failed to read following symlink, skipping", path, strerror(errno), "Warning");

}

symbuf[link] = '\0';

if((targetStat.st\_dev == newStat.st\_dev) && (targetStat.st\_ino == newStat.st\_ino)){

printf("%s SYMLINK RESOLVES TO TARGET\n", path);

}else{

if(targetStat.st\_size == newStat.st\_size)

{

sprintf(path, "%s/%s", dirName, symbuf);

if(checkFile(fileName, path) == 0){

printf("%s SYMLINK (%s) RESOLVES TO DUPLICATE\n", path, symbuf);

}

}

}

free(symbuf);

break;

}

break;

case DT\_LNK:

//dealing with symlinks

symbuf = malloc(newStat.st\_size + 1);

if((link = readlink(path, symbuf, newStat.st\_size +1)) == -1){

printError("Failed to read following symlink, skipping", path, strerror(errno), "Warning");

}

symbuf[link] = '\0';

if((targetStat.st\_dev == newStat.st\_dev) && (targetStat.st\_ino == newStat.st\_ino)){

printf("%s SYMLINK RESOLVES TO TARGET\n", path);

}else{

if(targetStat.st\_size == newStat.st\_size)

{

sprintf(path, "%s/%s", dirName, symbuf);

if(checkFile(fileName, path) == 0){

printf("%s SYMLINK (%s) RESOLVES TO DUPLICATE\n", path, symbuf);

}

}

}

free(symbuf);

break;

}

}

//error handling for failing to close directory

if(closedir(directory) == -1){

printError("Failed to close the directory!", dirName, strerror(errno), "Warning");

}

}

int main(int argc, char \*\*argv)

{

char \*targetFile = argv[1];

char \*targetDir = argv[2];

struct stat fileStat;

if(argc < 3){

printError("Not enough arguments! Use format [TargetFile] [TargetDirectory]\n", 0, 0, 0);

exit(1);

}

if(stat(targetFile, &fileStat) == -1){

printError("Not a valid target file!", targetFile, strerror(errno), "Fatal");

exit(1);

}

parseFile(targetDir, fileStat, targetFile, 1);

//everything ran with no errors

return 0;

}

Sample Run and Error Handling

