# CSCI 1730 Project 3

Differences in Files Using System Calls

## Learning Objectives

- Design and implement programs in a Unix environment that open, close, read, and write files.
- Demonstrate an understanding of the Unix file system by changing file permissions to allow programs to open, close, read, and write files.
- Design and implement a program that uses pointers and dynamic memory allocation and deallocation.
- Use valgrind to find memory leaks in programs. Implement programs that use dynamic memory allocation and deallocation and have no memory leaks.
- Use GDB to debug programs.

## Problem / Exercise

The goal of this project is to give you system programming experience with four file I/O system calls (open, close, read, and write), UNIX file permissions, and dynamic memory allocation and deallocation in C. In this project, you will implement a C program that reads in the contents of two input files. The program will then compare the two files, byte-by-byte, write differences into two output files, and output timings for the program's two main steps as stated in the project's requirements.

## Project Requirements

Your program must adhere to all requirements stated in this document.

1. Write a C program that compares two input text files and writes every byte in file one that is different from file two into a third file named `differencesFoundInFile1.txt` and every byte that is different in file two from file one into a fourth file named `differencesFoundInFile2.txt`. If `differencesFoundInFile1.txt` or `differencesFoundInFile2.txt` exists in the present working directory, then your program should overwrite the file(s) with new output (do not append to these files). If these output files do not exist in the present working directory, then your program should create these files and assign both of them read and write permissions to the owner. The C source code must be implemented in a single file called `proj3.c` and it must compile with our provided Makefile to create an executable called `proj3.out`. You cannot alter the Makefile provided. Place all files for this project in the same directory. Your program must use the system calls open, close, read, and write. Your program may use stat and printf, but it CANNOT use any other file I/O functions than those aforementioned.

2. Your program must complete the project requirement 1 in two different steps as indicated below.

Step 1: Compare input files one and two. This should be done by accessing 1 byte at a time from each of the files (use buffers of size 2: one byte to read and the other byte for a null character), comparing them, and writing any byte from file one that is not equal to the corresponding byte in file two into the file called `differencesFoundInFile1.txt`. This entire step should be implemented as a function called step1, and step1 should be called by your program's main function.

Step 2: Compare input files one and two. You will read both of the files into **two dynamically allocated arrays**. The input arrays should be allocated to be the **exact size needed** since we don't want to waste RAM. You will then compare the two arrays and write any byte from file two that is not equal to the corresponding byte in file one into a third array (also dynamically allocated), and then copy that third array into a file called `differencesFoundInFile2.txt`. This entire step should be implemented as a function called step2, and step2 should be called by your program's main function.

3. For each of the two steps in requirement 2, you will time how long the entire step takes to complete inside your program, and then output the two times to the command line when your program is finished (use `gettimeofday()`). In your README.txt file include the typical information as you did in previous projects and explain why the two times are different (if they are different). If they are not different, then explain why the two different steps results in the same times. Note: you will have to run larger input files to see a significant increase in the time the steps will take.

4. All arrays must be dynamically allocated and freed (with malloc/calloc and free). Programs that fail to do this will be penalized. Run valgrind to check for memory leaks. Your program should not contain any memory leaks. Your program must make at least one call to either malloc or calloc and at least one call to free.

5. Your program should be robust and include appropriate error checks. If there is an error reading a file, then your program should print (to standard output) "There was an error reading a file." If there is an error writing to a file, then your program should print (to standard output) "There was an error writing to a file." If there aren't enough command line arguments (or too many), then print the "Usage" message shown in the Examples section. After an error message is printed, then your program should terminate gracefully as discussed in lecture class.

6. You may ONLY use the following #include statements in your `proj3.c`. The use of any additional #include statements or source code copied/pasted (or included in your source code in any other way) from other libraries than those mentioned below will result in a failing grade on this project.

   (a) #include <fcntl.h>
   (b) #include <stdio.h>
   (c) #include <stdlib.h>
   (d) #include <sys/stat.h>
   (e) #include <sys/time.h>
   (f) #include <sys/types.h>
   (g) #include <unistd.h>

7. You may implement additional C functions and call upon these C functions that you implemented as needed to complete this project.

8. Do not use any global variables. Do not use any global pointers. Violating any one of these rules will result in a grade of zero on this assignment.

9. Do not use any static variables. Do not use any static pointers. Violating any one of these rules will result in a grade of zero on this assignment.

10. If you encounter any runtime or logic errors while implementing your project, then use GDB to debug your program.

11. Your final program's executable, `proj3.out`, must have I/O that matches the examples exactly except for the timing output, which will vary each time your program is executed. Also, your program must correctly create (or overwrite) the two files (with read and write permissions for the owner) as aforementioned. Failure to follow the I/O provided in the Examples section may result in a failing grade for this project.

## Coding Style Requirements

1. All functions must be commented. Comments must include a brief description of what the function does, its input(s), its output, and any assumptions associated with calling it. If your function has a prototype and an implementation separated into different parts of your source code, then you only need to put the comments for that function above its prototype (there is no need to comment both the prototype and implementation; commenting the prototype is sufficient).

2. All structs, unions, and enums must be commented.

3. All identifiers must be named well to denote their functionality. Badly named identifiers are not allowed. For example, identifiers like a, aaa, b, bbb, bbbb are bad names for identifiers.

4. Every line of source code must be indented properly and consistently.

## README.txt File Requirements

Make sure to include a `README.txt` file (use a .txt file extension) that includes the following information presented in a reasonably formatted way:

- Your First and Last Name (as they appear on eLC) and your 810/811#
- Instructions on how to compile and run your program. Since we are using a Makefile for this assignment, then these instructions should pretty simple based on the provided Makefile.
- Explain why or why not the times for step 1 and step 2 are different as aforementioned.

## Examples

The I/O of your final program must match the examples except for the timings, which will vary each time your program is executed. The input files can be found on eLC in examples.tar.gz. The entire file, examples.tar.gz, should be uploaded to your odin account, and its contents should be extracted using the `tar` command below on odin.

```
tar -xvf examples.tar.gz
```

The first line of each example contains one or more commands (separated by a ; for multiple commands), and your program should produce the correct output along with the two output files mentioned in the project's requirements. You must check that all characters and character counts matches these examples exactly (otherwise, you may fail this project).

```
./proj3.out
Usage: proj3.out <file1> <file2>

./proj3.out input1.txt
Usage: proj3.out <file1> <file2>

./proj3.out a b
There was an error reading a file.

./proj3.out input1.txt input2.txt; cat differencesFoundInFile1.txt differencesFoundInFile2.txt;
Step 1 took 0.038000 milliseconds
Step 2 took 0.018000 milliseconds
cat!dog.

./proj3.out input1.txt input2.txt; cat differencesFoundInFile1.txt differencesFoundInFile2.txt | wc -c;
Step 1 took 0.066000 milliseconds
Step 2 took 0.040000 milliseconds
8

./proj3.out input3.txt input4.txt; cat differencesFoundInFile1.txt differencesFoundInFile2.txt;
Step 1 took 0.034000 milliseconds
Step 2 took 0.016000 milliseconds
aaaaa
b

./proj3.out input3.txt input4.txt; cat differencesFoundInFile1.txt differencesFoundInFile2.txt | wc -c;
Step 1 took 0.056000 milliseconds
Step 2 took 0.017000 milliseconds
8

./proj3.out input5.txt input6.txt; cat differencesFoundInFile1.txt differencesFoundInFile2.txt;
Step 1 took 0.027000 milliseconds
Step 2 took 0.039000 milliseconds
a
bbbbbb
```

```
./proj3.out input5.txt input6.txt; cat differencesFoundInFile1.txt differencesFoundInFile2.txt | wc -c;
Step 1 took 0.026000 milliseconds
Step 2 took 0.016000 milliseconds
9

./proj3.out alice1.txt alice2.txt; cat differencesFoundInFile1.txt differencesFoundInFile2.txt;
Step 1 took 113.053000 milliseconds
Step 2 took 0.724000 milliseconds
showingAliceBender*Zoey!

./proj3.out alice1.txt alice2.txt; cat differencesFoundInFile1.txt differencesFoundInFile2.txt | wc -c;
Step 1 took 113.401000 milliseconds
Step 2 took 0.784000 milliseconds
24

./proj3.out alice1.txt alice1.txt; cat differencesFoundInFile1.txt differencesFoundInFile2.txt | wc -c;
Step 1 took 114.963000 milliseconds
Step 2 took 0.646000 milliseconds
0
```

## Submission

Submit your files before the due date and due time stated on eLC. Submit your files on eLC under the Assignment named Project 3. Submit only the following files.

1. All source files required for this assignment: proj3.c
2. A README.txt file filled out correctly

Do not submit any compiled code. Also, do not submit any zipped files or directories. Do not submit a Makefile. We will use our own, unmodified copy of the provided Makefile to compile your program. We only need the files mentioned above with the file extensions aforementioned.

# 1 Grading (30 points)

If your program does not compile on odin using the provided Makefile and using the required compiler for this course (`gcc version 11.2.0`), then you'll receive a grade of zero for this assignment. Otherwise, your program will be graded using the criteria below.

| | |
|---|---|
| Program runs correctly with various test cases on odin | 30 points |
| README.txt file missing or incorrect in some way | -3 points |
| Not submitting to the correct Assignment on eLC | -3 points |
| Late penalty for submitting 0 hours to 24 hours late | -6 points |
| One or more compiler warnings | -6 points |
| Not adhering to one or more coding style requirements | -6 points |
| Valgrind identifies a memory leak (definitely, indirectly, or possibly lost) | -6 points |
| Source code contains an unauthorized include statement | -30 points |
| Source code contains an unauthorized file I/O function call | -30 points |
| Source code contains a global variable or global pointer | -30 points |
| Source code contains a static variable or static pointer | -30 points |
| Penalty for not following instructions (invalid I/O, etc.) | Penalty decided by grader |

You must test, test, and retest your code to make sure it compiles and runs correctly on odin with any given set of inputs. This means that you need to create many examples on your own (that are different than the aforementioned examples) to ensure you have a correctly working program. Your program's I/O must match the examples given in the Examples section except for the timings, which will vary each time your program is executed. You may NOT assume inputs will be valid.