**GEBZE TECHNICAL UNIVERSITY**

**COMPUTER ENGINEERING**

**DEPARTMENT**

**CSE344 Systems Programming**

**Spring 2025**

**Homework 1 Report**

**Osmancan Bozali**

**220104004011**

## **Introduction:**

This report presents the implementation of a Secure File and Directory Management System developed in C for CSE344 Systems Programming course. The program utilizes Linux system calls to perform file and directory operations, including creation, listing, reading, updating, and deletion, while incorporating process creation with fork() and logging functionality. The system ensures secure file operations through file locking. This assignment was tested on Debian 11 (64-bit) in VirtualBox on my machine, adhering to the provided requirements. The purpose of this homework is to simulate a basic file management utility and operation logging.

## **Code Explanation:**

The program has several functions, each is written to handle a specific file or directory operation (or helper function for operations). Below is an explanation of the functions and their roles (see the source code for full function implementations):

**writeMessage(int fd, const char\* prefix, const char\* name, const char\* suffix)**: This utility function writes a formatted message to a specified file descriptor. It surrounds the name parameter with quotes, ensuring consistent error message formatting (e.g., Error: File "example.txt" not found). It’s used for user feedback and aligns with the assignment’s output requirements.

**getTimestamp()**: Generates a timestamp in the format [YYYY-MM-DD HH:MM:SS] using time() and strftime(). It returns a string, which is used for logging operations and writing creation timestamps into new files.

**logOperationToFile(const char\* operation, const char\* name, const char\* result)**: Appends a log entry to log.txt with a timestamp, operation type, target name (if applicable), and result. It uses open() with O\_APPEND to ensure logs are cumulative.

**createDirectory(const char \*name)**: Creates a directory using mkdir() with permissions 0777. If the directory exists (EEXIST), it outputs an error and logs the failure. Otherwise, it logs success. This implements the createDir command with proper error handling.

**createFile(const char \*name)**: Creates a file using open() with O\_CREAT | O\_EXCL to prevent overwriting existing files. On success, it writes the current timestamp to the file. Errors (e.g., file exists) trigger appropriate messages and logs.

**listDirectory(const char \*name)**: Uses fork() to create a child process that lists all files in the directory with opendir() and readdir(). The child exits with status 0 on success or 1 if the directory isn’t found, and the parent logs the outcome.

**listFilesByExtension(const char \*name, const char \*extension)**: Similar to listDirectory(), but filters files by extension using strstr(). It uses fork() and counts matching files; if none are found, it prints a specific message (e.g., No file with extension ".txt" found in "testDir"). The operation is logged at the end.

**readFile(const char \*name)**: Opens a file with O\_RDONLY and reads its content into a buffer, printing it to stdout using write(). If the file doesn’t exist, it outputs an error and logs the failure.

**appendToFile(const char \*name, const char \*content)**: Opens a file with O\_APPEND and uses flock(LOCK\_EX | LOCK\_NB) to lock it exclusively, preventing simultaneous writes. If locked or inaccessible, it reports an error; otherwise, it appends the content and logs success.

**deleteFile(const char \*name)**: Uses fork() to delete a file with unlink() in a child process. The parent waits and logs success or failure (e.g., file not found). This implements deleteFile with process separation as required.

**deleteDirectory(const char \*name)**: Uses fork() to delete an empty directory with rmdir() in a child process. It handles errors like non-empty directories (ENOTEMPTY) or non-existent directories (ENOENT), logging the result.

**showLogs()**:  
Reads and displays the contents of log.txt using read() and write(). If the log file is missing, it reports an error. This operation is logged in the log.txt too.

**showHelp()**:  
Prints a usage guide with all commands and descriptions when no arguments are provided (e.g., Usage: fileManager <command> [arguments]). It logs the display action.

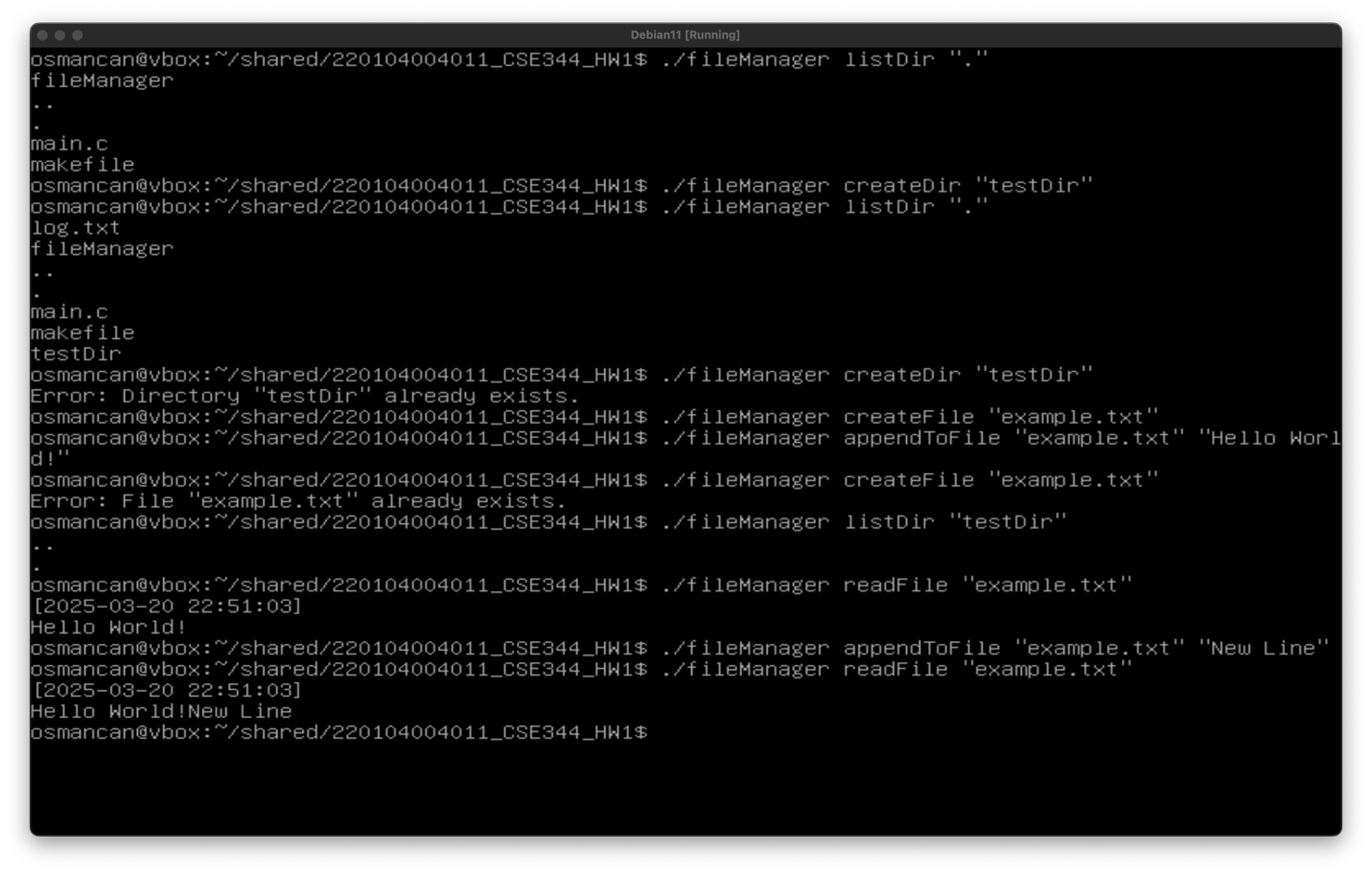
**main(int argc, char \*argv[])**:  
The entry point parses command-line arguments and calls the appropriate function based on the command (e.g., createDir, showLogs). It checks for missing arguments and triggers showHelp() if no command is given, ensuring robust command handling.

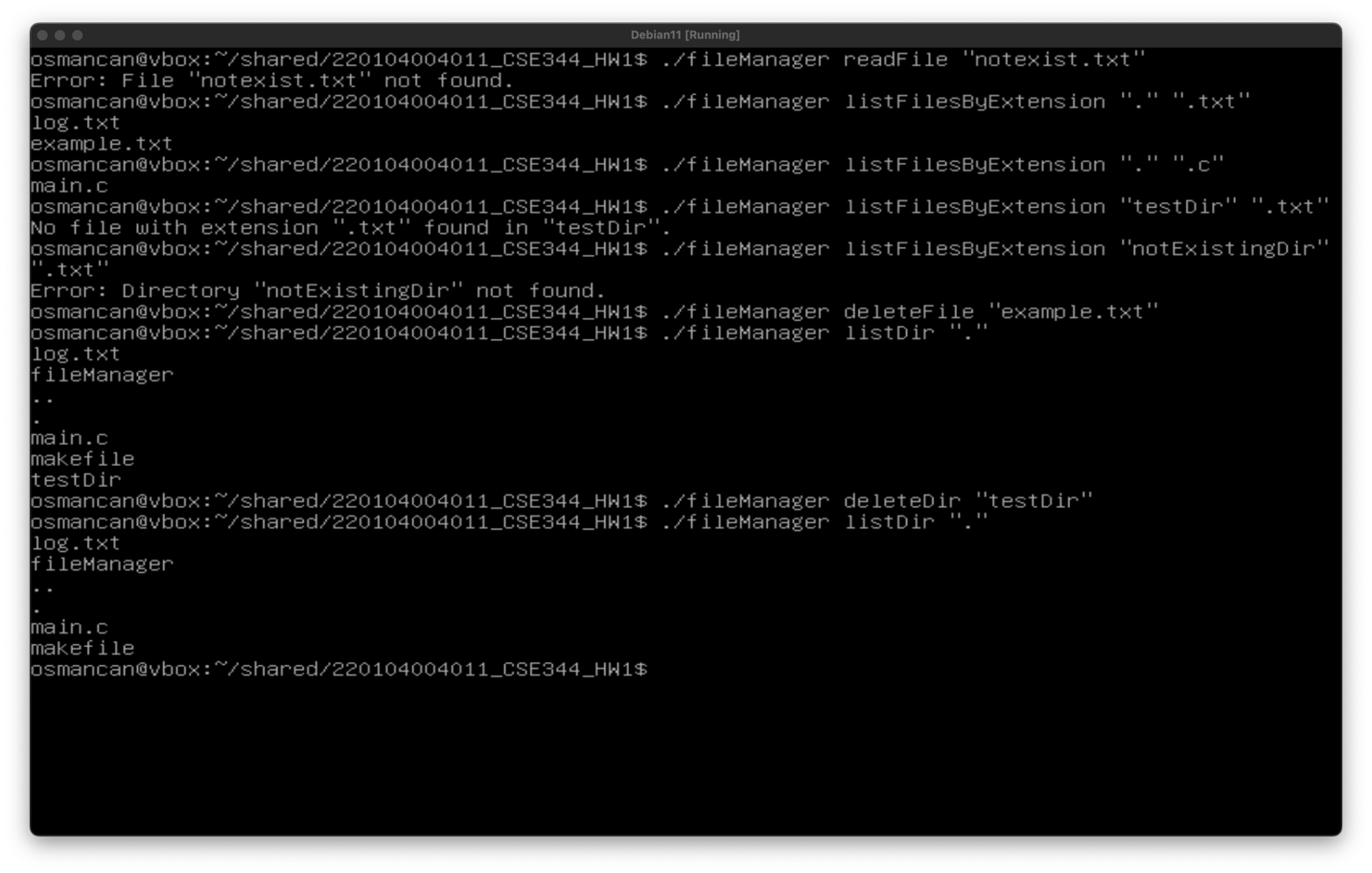
Each function uses system calls like open(), write(), and fork() to interact with the file system securely and logs operations consistently, adhering to the assignment’s specifications.

## **Tests and Screenshots:**

In order to test the program yourself, you can use the makefile

* make : This command cleans the all related files/folders and compiles the program (preferred, just run this command and everything will be ready)
* make clean : This command clears all the files/folders created while running the program)

listDir, createDir, createFile, appendToFile, readFile function tests with edge cases:

readFile, listFilesByExtension, deleteFile, listDir, deleteDir function tests with edge cases:

A screenshot of a computer

AI-generated content may be incorrect.showLogs function test (logs of previous test results above):

**A screenshot of a computer

AI-generated content may be incorrect.**deleteDir with non-empty directory and showHelp function tests:

## 

## **Conclusion:**

This homework was a great exercise in understanding Linux system calls, process management with fork(), and file locking with flock().While implementing the homework I made research on the web consistently since I am a beginner in terms of Systems Programming. One challenge was ensuring proper error handling for all edge cases, such as existing files or locked resources, which I solved with errno and it’s error codes. Another difficulty was managing child processes effectively led me to use waitpid() to synchronize parent and child execution. Alse the string formatting was a bit challenging. Overall, the implementation meets all requirements, and the testing scenario added as a proof. By implementing this homework, I think I made a good introduction to Systems Programming.