CSE 344 Homework #1 Report

Secure File and Directory Management System

Student Name: Recep Furkan Akın

Student ID: 210104004042

Course: CSE 344

Homework: #1

---

1. Introduction

The File Management System is a command-line utility built in C that provides basic file and directory operations. It allows users to create, read, modify, and delete files and directories, all while implementing proper file locking mechanisms to prevent concurrent access issues. The program uses system calls directly rather than standard library functions, demonstrating low-level file I/O operations.

---

2. Code Explanation

fileManager.c

- **createDirectory** :

The `createDirectory` function is all about making a new folder with a particular name (`dirName`). Here's how it works: First, it checks if the folder is already there by using the `stat` system call. This call gets information about the file or directory named `dirName`. If `stat` returns `-1`, it means the folder isn't there yet, so the function goes ahead and creates it.

To make the folder, the `mkdir` system call is used, setting permissions to `0755`. This permission lets the owner do everything—read, write, and execute—but others can only read and execute. If the `mkdir` call doesn't work (returns `-1`), an error message is made using `strerror(errno)` to get the error description, and this message is written to the standard error output, or `STDERR_FILENO`. The function stops early by using `return`.

When the folder is created successfully, a success message is made and logged using the `logOperation` function, and this message is also sent to the standard output (`STDOUT_FILENO`) to let the user know.

If the `stat` call finds that the folder already exists, an error message is made to let the user know that the folder can't be created because it's already there. This message is also sent to the standard error output.

- **`createFile`** :

The `createFile` function is crafted to generate a new file with a given name, called `fileName`. To start, it checks if the file is already there using the `stat` system call. The `stat` function grabs details about the file, and when it returns `-1`, it shows the file doesn't exist, letting the function move forward with creating the file.

For creating the file, the `open` system call is used with these flags: `O_WRONLY | O_CREAT | O_TRUNC` . These flags mean the file should be opened for writing, created if it's not there, and shortened to zero length if it is there. The file permissions are set to `0644`, which means the owner can read and write, while others can only read. If the open call doesn't work (returns `-1`), an error message is made using `strerror(errno)` to get the error description, and the message is sent to the standard error output (`STDERR_FILENO`). The function then stops early using `return`.

If the file gets created successfully, the function writes a timestamp and a "File created" message into the file. The timestamp is made using the `getCurrentTimestamp` function, which is thought to fill the `timestamp` buffer with the current date and time. The `write` system call is used to write this timestamp and message into the file. After writing, the file descriptor is closed using the `close` system call to free up system resources.

A success message is then created and logged using the `logOperation` function. This message is also sent to the standard output (`STDOUT_FILENO`) to let the user know that the file got created successfully.

If the `stat` call finds out that the file already exists, an error message is made to let the user know that the file can't be created because it already exists. This message is sent to the standard error output.

- **`listDirectory`**:

The `listDirectory` function is designed to list the contents of a specified directory (`dirName`). It begins by checking if the directory exists using the `stat` system call. If the `stat` call fails (returns `-1`), it constructs an error message indicating that the directory was not found and writes this message to the standard error output (`STDERR_FILENO`). The function then exits early.

Next, the function creates a new process using the `fork` system call. If `fork` fails (returns `-1`), an error message is constructed and written to the standard error output, and the function exits. If the `fork` call succeeds, the process is split into a parent process and a child process.

In the child process (`pid == 0`), the function attempts to open the directory using the `opendir` function. If the directory cannot be opened, an error message is written to the standard error output, and the child process exits with a failure status. If the directory is successfully opened, the function iterates through its entries using `readdir`. For each entry, it constructs the full path of the entry and retrieves its metadata using the `stat` system call. Based on the file type (directory, regular file, or other), it categorizes the entry and constructs a message describing the entry's name and type. This message is written to the standard output (`STDOUT_FILENO`). After processing all entries, the directory is closed using `closedir`, and the child process exits successfully.

In the parent process, the function waits for the child process to complete using `waitpid`. Once the child process finishes, the parent constructs a log message indicating that the directory was listed successfully. This message is logged using the `logOperation` function.

- `listFilesByExtension` :

The `listFilesByExtension` function is designed to list all files in a specified directory (`dirName`) that have a specific file extension (`extension`). It begins by checking if the directory exists using the `stat` system call. If the directory does not exist (`stat` returns `-1`), an error message is constructed and written to the standard error output (`STDERR_FILENO`), and the function exits early.

The function then creates a new process using the `fork` system call. If `fork` fails (returns `-1`), an error message is written to the standard error output, and the function exits. If the `fork` call succeeds, the process is split into a parent process and a child process.

In the child process (`pid == 0`), the function attempts to open the directory using the `opendir` function. If the directory cannot be opened, an error message is written to the standard error output, and the child process exits with a failure status. If the directory is successfully opened, the function iterates through its entries using `readdir`. For each entry, it checks if the file name contains the specified extension by using `strrchr` to locate the last occurrence of a period (`.`) in the file name and comparing the substring after the period with the given extension using `strcmp`. If a match is found, the file is considered to have the desired extension.

For each matching file, the function constructs a message containing the file name and writes it to the standard output (`STDOUT_FILENO`). If no files with the specified extension are found, a message indicating this is written to the standard output. After processing all entries, the directory is closed using `closedir`, and the child process exits successfully.

In the parent process, the function waits for the child process to complete using `waitpid`. Once the child process finishes, the parent constructs a log message indicating that the files with the specified extension were listed successfully. This message is logged using the `logOperation` function.

- **`readFile`**:

This `readFile` function uses file locking for concurrent access control. It first checks if the file exists using `stat()` and displays an error if it doesn't. It then attempts to open the file in read-only mode and displays an error if it fails.

Next, it applies a shared lock using `flock(fd, LOCK_SH)` to allow multiple readers but prevent writers from modifying the file while being read. If locking fails, it displays an error, closes the file, and returns.

Once the file is locked, it reads its contents in chunks using a buffer of size `BUFFER_SIZE`. It displays a header indicating the file being read and enters a loop that reads chunks from the file and writes them directly to standard output. The `read()` function returns the number of bytes read, which is used to ensure we write exactly that many bytes to stdout.

After reading, the function releases the lock, closes the file descriptor, and logs the successful read operation using `logOperation()`.

This implementation demonstrates proper file handling practices: checking for errors, using appropriate locking, cleaning up resources, and providing detailed error messages.

- **`appendToFile`:**

The `appendToFile` function ensures thread-safe file modification using advisory locks. It begins by checking if the target file exists via the `stat` system call. If the file is not found, the function immediately returns an error to `STDERR`, notifying the user that the file is missing. If the file exists, it opens it in append mode (`O_WRONLY | O_APPEND`) to guarantee that writes occur at the end of the file. If opening fails—due to issues like read-only permissions or a locked state—the function writes an error to `STDERR` and exits.

Next, the function applies an **exclusive lock** using `flock(fd, LOCK_EX)`, which blocks other processes from accessing the file until the lock is released. If locking fails (e.g., due to an unresolvable conflict), the function writes an error, closes the file descriptor, and exits. Once the lock is acquired, the function appends the provided `content` to the file, prefixed by a newline (`\n`), ensuring atomicity through the lock.

After writing, the lock is explicitly released with `flock(fd, LOCK_UN)`, and the file descriptor is closed. Finally, the function logs the successful operation to `log.txt` via `logOperation` and prints a confirmation message to `STDOUT`.

- **`deleteFile`:**

The `deleteFile` function is designed to delete a specified file (`fileName`) from the file system. It begins by checking if the file exists using the `stat` system call. If the file does not exist (`stat` returns `-1`), an error message is constructed and written to the standard error output (`STDERR_FILENO`), and the function exits early.

If the file exists, the function creates a new process using the `fork` system call. This approach isolates the deletion operation in a child process, ensuring that any failures during deletion do not affect the parent process. If the `fork` call fails (returns `-1`), an error message is constructed using `strerror(errno)` to retrieve the error description, and the message is written to the standard error output. The function then exits without proceeding further.

In the child process (`pid == 0`), the function attempts to delete the file using the `unlink` system call. The `unlink` function removes the specified file from the file

system. If the `unlink` call fails (returns `-1`), an error message is constructed and written to the standard error output, and the child process exits with a failure status. If the file is successfully deleted, a success message is written to the standard output (`STDOUT_FILENO`) to inform the user, and the child process exits with a success status.

In the parent process, the function waits for the child process to complete using `waitpid`. Once the child process finishes, the parent checks if it exited normally (`WIFEXITED(status)`) and with a success status (`WEXITSTATUS(status) == EXIT_SUCCESS`). If both conditions are met, the parent constructs a log message indicating that the file was deleted successfully and logs this message using the `logOperation` function.

- **`deleteDirectory`** :

The `deleteDirectory` function is designed to remove a directory from the file system, but only if it exists and is empty. It starts with a check using the `stat` system call to verify if the directory exists. If the directory doesn't exist (indicated by `stat` returning `-1`), an error message is constructed and sent to the standard error stream (`STDERR_FILENO`), and the function returns early.

The function then creates a new process using `fork()` to isolate the directory deletion operation. If the fork fails, an error message is generated using `strerror(errno)` to provide context about the failure, and the function returns.

In the child process, the function first attempts to open the directory with `opendir()`. If this fails, an error message is displayed, and the child process exits with a failure status. If successful, the function then checks if the directory is empty by reading its entries with `readdir()` in a loop. It ignores the special entries "." (current directory) and ".." (parent directory) and looks for any other entries. If any other entry is found, the directory is considered not empty.

After determining if the directory is empty (by closing the directory handle with `closedir()`), the function handles two possible scenarios: If the directory is not empty, it displays an error message and exits with a failure status. If the directory is empty, it attempts to remove it using the `rmdir()` system call. If `rmdir()` fails, an error message is displayed, and the function exits with a failure status. If the directory is successfully deleted, a success message is sent to standard output, and the child process exits with a success status.

Meanwhile, the parent process waits for the child to complete using `waitpid()`. It then checks the child's exit status using the macros `WIFEXITED` (to verify normal termination) and `WEXITSTATUS` (to check the exit code). If the child exited normally with a success status, the parent logs the successful deletion operation using the `logOperation()` function.

- **`showLogs`** :

The `showLogs` function is designed to display the contents of a log file to the user via standard output. It begins by verifying whether the log file (defined by the constant `LOG_FILE`) exists on the file system using the `stat` system call. If the file doesn't exist (indicated by `stat` returning `-1`), an error message is sent to the standard error stream (`STDERR_FILENO`), and the function returns early without attempting to read a non-existent file.

If the log file exists, the function tries to open it in read-only mode using the `open` system call with the `O_RDONLY` flag. Should the file opening operation fail for any reason (such as insufficient permissions), the function generates an error message that includes the specific error description obtained from `strerror(errno)`. This error message is written to standard error, and the function returns without further processing.

Once the log file is successfully opened, the function prepares to read its contents. It declares a buffer array to store chunks of data read from the file and a variable to track the number of bytes read in each operation. Before displaying the file contents, the function writes a header message ("Operation logs:") to standard output to provide context for the information that follows.

The function then enters a loop where it reads data from the file in chunks up to the size of the buffer using the `read` system call. Each successfully read chunk is immediately written to standard output using the `write` system call. This process continues until `read` returns a value less than or equal to zero, indicating either the end of the file or an error.

After all file contents have been displayed, the function properly closes the file descriptor with the `close` system call to release system resources. Finally, the function logs its own activity by calling the `logOperation` function with a message indicating that the logs were successfully displayed.

utils.c

- **`logOperation`**:

The `logOperation` function writes log entries to a file with timestamps. It opens the log file with write, create, and append modes. If opening fails, it outputs an error message to standard error. Otherwise, it gets the current timestamp, writes the timestamp followed by the provided message to the log file, adds a newline, and closes the file. The function uses low-level file operations (`open`, `write`, `close`) rather than higher-level I/O functions.

- **`getCurrentTimestamp`**:

The `getCurrentTimestamp` function creates a formatted date-time string. It obtains the current time with `time(NULL)`, converts it to a structured format with `localtime()`, and then formats it into a human-readable string "[YYYY-MM-DD HH:MM:SS]" using `strftime()`. This formatted timestamp is stored in the pre-allocated buffer passed as a parameter, making it ready for use in logs or display output.

- displayHelp:

The `displayHelp` function provides usage instructions for the fileManager utility. It defines a multi-line string with the command syntax and options, then writes this help text directly to standard output using the low-level `write` system call rather than using printf. The function lists all available commands with their required arguments and brief descriptions, giving users a complete reference for the program's functionality.

`main.c`

- Parses CLI arguments and routes to corresponding functions.

---

3. Screenshots

Simple Test Scenario Execution (mentioned in PDF)

```
recepfurkanakin@system:~/SystemHW25/hw1$ make test_simple
Running simple test scenario...
1. Creating directory testDir
./fileManager createDir testDir
Directory "testDir" created successfully.

2. Creating file testDir/example.txt
./fileManager createFile testDir/example.txt
File "testDir/example.txt" created successfully.

3. Appending initial content to file
./fileManager appendToFile testDir/example.txt "Hello, World!"
Content appended to file "testDir/example.txt" successfully.

4. Listing directory contents
./fileManager listDir testDir
Contents of directory "testDir":
  .. [Directory]
  example.txt [File]
  . [Directory]

5. Reading file content
./fileManager readFile testDir/example.txt
Contents of file "testDir/example.txt":
[2025-03-22 22:32:36] File created

Hello, World!

6. Appending more content to file
./fileManager appendToFile testDir/example.txt "New Line"
Content appended to file "testDir/example.txt" successfully.

7. Reading updated file content
./fileManager readFile testDir/example.txt
Contents of file "testDir/example.txt":
[2025-03-22 22:32:36] File created

Hello, World!
New Line

8. Deleting file
./fileManager deleteFile testDir/example.txt
File "testDir/example.txt" deleted successfully.

9. Showing logs
./fileManager showLogs
Operation logs:
[2025-03-22 22:32:36] Directory "testDir" created successfully.
[2025-03-22 22:32:36] File "testDir/example.txt" created successfully.
[2025-03-22 22:32:36] Content appended to file "testDir/example.txt" successfully.
[2025-03-22 22:32:36] Directory "testDir" listed successfully.
[2025-03-22 22:32:36] File "testDir/example.txt" read successfully.
[2025-03-22 22:32:36] Content appended to file "testDir/example.txt" successfully.
[2025-03-22 22:32:36] File "testDir/example.txt" read successfully.
[2025-03-22 22:32:36] File "testDir/example.txt" deleted successfully.

Simple test scenario completed successfully!
```

```
recepfurkanakin@system:~/SystemHW25/hw1$ ./fileManager
Usage: fileManager <command> [arguments]
Commands:
  createDir "folderName" - Create a new directory
  createFile "fileName" - Create a new file
  listDir "folderName" - List all files in a directory
  listFilesByExtension "folderName" ".txt" - List files with specific extension
  readFile "fileName" - Read a file's content
  appendToFile "fileName" "new content" - Append content to a file
  deleteFile "fileName" - Delete a file
  deleteDir "folderName" - Delete an empty directory
  showLogs - Display operation logs
```

Comprehensive Test Scenario (all functions tested)

```
recepfurkanakin@system:~/SystemHW25/hw1$ make comprehensive_test

===== TESTING SUCCESSFUL OPERATIONS =====
1. Creating test directory structure
./fileManager createDir test_dir
Directory "test_dir" created successfully.
./fileManager createDir test_dir/subdir
Directory "test_dir/subdir" created successfully.

2. Creating various files
./fileManager createFile test_dir/file1.txt
File "test_dir/file1.txt" created successfully.
./fileManager createFile test_dir/file2.txt
File "test_dir/file2.txt" created successfully.
./fileManager createFile test_dir/document.doc
File "test_dir/document.doc" created successfully.
./fileManager createFile test_dir/program.c
File "test_dir/program.c" created successfully.
./fileManager createFile test_dir/subdir/nested.txt
File "test_dir/subdir/nested.txt" created successfully.

3. Testing listing operations
./fileManager listDir test_dir
Contents of directory "test_dir":
  document.doc [File]
  file1.txt [File]
  .. [Directory]
  subdir [Directory]
  file2.txt [File]
  . [Directory]
  program.c [File]
./fileManager listDir test_dir/subdir
Contents of directory "test_dir/subdir":
  .. [Directory]
  . [Directory]
  nested.txt [File]
./fileManager listFilesByExtension test_dir .txt
Files with extension ".txt" in directory "test_dir":
  file1.txt
  file2.txt
./fileManager listFilesByExtension test_dir .c
Files with extension ".c" in directory "test_dir":
  program.c

4. Testing file content operations
./fileManager appendToFile test_dir/file1.txt "First line of content"
Content appended to file "test_dir/file1.txt" successfully.
./fileManager appendToFile test_dir/file1.txt "Second line of content"
Content appended to file "test_dir/file1.txt" successfully.
./fileManager readFile test_dir/file1.txt
Contents of file "test_dir/file1.txt":
[2025-03-22 22:53:39] File created

First line of content
Second line of content
```

```
5. Testing file deletion
./fileManager deleteFile test_dir/program.c
File "test_dir/program.c" deleted successfully.
./fileManager listDir test_dir
Contents of directory "test_dir":
  document.doc [File]
  file1.txt [File]
  .. [Directory]
  subdir [Directory]
  file2.txt [File]
  . [Directory]

6. Testing logs
./fileManager showLogs
Operation logs:
[2025-03-22 22:53:39] Directory "test_dir" created successfully.
[2025-03-22 22:53:39] Directory "test_dir/subdir" created successfully.
[2025-03-22 22:53:39] File "test_dir/file1.txt" created successfully.
[2025-03-22 22:53:39] File "test_dir/file2.txt" created successfully.
[2025-03-22 22:53:39] File "test_dir/document.doc" created successfully.
[2025-03-22 22:53:39] File "test_dir/program.c" created successfully.
[2025-03-22 22:53:39] File "test_dir/subdir/nested.txt" created successfully.
[2025-03-22 22:53:39] Directory "test_dir" listed successfully.
[2025-03-22 22:53:39] Directory "test_dir/subdir" listed successfully.
[2025-03-22 22:53:39] Listed files with extension ".txt" in directory "test_dir".
[2025-03-22 22:53:39] Listed files with extension ".c" in directory "test_dir".
[2025-03-22 22:53:39] Content appended to file "test_dir/file1.txt" successfully.
[2025-03-22 22:53:39] Content appended to file "test_dir/file1.txt" successfully.
[2025-03-22 22:53:39] File "test_dir/file1.txt" read successfully.
[2025-03-22 22:53:39] File "test_dir/program.c" deleted successfully.
[2025-03-22 22:53:39] Directory "test_dir" listed successfully.

===== SUCCESS TESTS COMPLETED =====
```

```
===== TESTING ERROR CONDITIONS =====
./fileManager createDir test_error_dir
Directory "test_error_dir" created successfully.

1. Testing directory errors
   a. Creating directory that already exists
./fileManager createDir test_error_dir
Error: Directory "test_error_dir" already exists.
   b. Listing non-existent directory
./fileManager listDir non_existent_dir
Error: Directory "non_existent_dir" not found.
   c. Listing directory with wrong params
./fileManager listDir
Usage: fileManager <command> [arguments]
Commands:
  createDir "folderName" - Create a new directory
  createFile "fileName" - Create a new file
  listDir "folderName" - List all files in a directory
  listFilesByExtension "folderName" ".txt" - List files with specific extension
  readFile "fileName" - Read a file's content
  appendToFile "fileName" "new content" - Append content to a file
  deleteFile "fileName" - Delete a file
  deleteDir "folderName" - Delete an empty directory
  showLogs - Display operation logs

2. Testing file errors
   a. Creating file in non-existent directory
./fileManager createFile non_existent_dir/file.txt
Error creating file: No such file or directory
   b. Creating file that already exists
./fileManager createFile test_error_dir/duplicate.txt
File "test_error_dir/duplicate.txt" created successfully.
./fileManager createFile test_error_dir/duplicate.txt
Error: File "test_error_dir/duplicate.txt" already exists.
   c. Reading non-existent file
./fileManager readFile test_error_dir/missing.txt
Error: File "test_error_dir/missing.txt" not found.
   d. Reading with wrong params
./fileManager readFile
Usage: fileManager <command> [arguments]
Commands:
  createDir "folderName" - Create a new directory
  createFile "fileName" - Create a new file
  listDir "folderName" - List all files in a directory
  listFilesByExtension "folderName" ".txt" - List files with specific extension
  readFile "fileName" - Read a file's content
  appendToFile "fileName" "new content" - Append content to a file
  deleteFile "fileName" - Delete a file
  deleteDir "folderName" - Delete an empty directory
  showLogs - Display operation logs
```

```
3. Testing append errors
   a. Appending to non-existent file
./fileManager appendToFile non_existent_file.txt "Some content"
Error: File "non_existent_file.txt" not found.
   b. Appending with wrong params
./fileManager appendToFile
Usage: fileManager <command> [arguments]
Commands:
  createDir "folderName" - Create a new directory
  createFile "fileName" - Create a new file
  listDir "folderName" - List all files in a directory
  listFilesByExtension "folderName" ".txt" - List files with specific extension
  readFile "fileName" - Read a file's content
  appendToFile "fileName" "new content" - Append content to a file
  deleteFile "fileName" - Delete a file
  deleteDir "folderName" - Delete an empty directory
  showLogs - Display operation logs
./fileManager appendToFile test_error_dir/some_file.txt
Usage: fileManager <command> [arguments]
Commands:
  createDir "folderName" - Create a new directory
  createFile "fileName" - Create a new file
  listDir "folderName" - List all files in a directory
  listFilesByExtension "folderName" ".txt" - List files with specific extension
  readFile "fileName" - Read a file's content
  appendToFile "fileName" "new content" - Append content to a file
  deleteFile "fileName" - Delete a file
  deleteDir "folderName" - Delete an empty directory
  showLogs - Display operation logs

4. Testing delete errors
   a. Deleting non-existent file
./fileManager deleteFile test_error_dir/non_existent.txt
Error: File "test_error_dir/non_existent.txt" not found.
   b. Deleting with wrong params
./fileManager deleteFile
Usage: fileManager <command> [arguments]
Commands:
  createDir "folderName" - Create a new directory
  createFile "fileName" - Create a new file
  listDir "folderName" - List all files in a directory
  listFilesByExtension "folderName" ".txt" - List files with specific extension
  readFile "fileName" - Read a file's content
  appendToFile "fileName" "new content" - Append content to a file
  deleteFile "fileName" - Delete a file
  deleteDir "folderName" - Delete an empty directory
  showLogs - Display operation logs
```

```
5. Testing extension filtering errors
   a. Extension filtering with non-existent dir
./fileManager listFilesByExtension non_existent_dir .txt
Error: Directory "non_existent_dir" not found.
   b. Extension filtering with wrong params
./fileManager listFilesByExtension
Usage: fileManager <command> [arguments]
Commands:
  createDir "folderName" - Create a new directory
  createFile "fileName" - Create a new file
  listDir "folderName" - List all files in a directory
  listFilesByExtension "folderName" ".txt" - List files with specific extension
  readFile "fileName" - Read a file's content
  appendToFile "fileName" "new content" - Append content to a file
  deleteFile "fileName" - Delete a file
  deleteDir "folderName" - Delete an empty directory
  showLogs - Display operation logs
./fileManager listFilesByExtension test_error_dir
Usage: fileManager <command> [arguments]
Commands:
  createDir "folderName" - Create a new directory
  createFile "fileName" - Create a new file
  listDir "folderName" - List all files in a directory
  listFilesByExtension "folderName" ".txt" - List files with specific extension
  readFile "fileName" - Read a file's content
  appendToFile "fileName" "new content" - Append content to a file
  deleteFile "fileName" - Delete a file
  deleteDir "folderName" - Delete an empty directory
  showLogs - Display operation logs

6. Testing invalid commands
./fileManager invalidCommand test_error_dir
Usage: fileManager <command> [arguments]
Commands:
  createDir "folderName" - Create a new directory
  createFile "fileName" - Create a new file
  listDir "folderName" - List all files in a directory
  listFilesByExtension "folderName" ".txt" - List files with specific extension
  readFile "fileName" - Read a file's content
  appendToFile "fileName" "new content" - Append content to a file
  deleteFile "fileName" - Delete a file
  deleteDir "folderName" - Delete an empty directory
  showLogs - Display operation logs

===== ERROR TESTS COMPLETED =====

===== ALL TESTS COMPLETED SUCCESSFULLY
```

```
========== TESTING FILE LOCKING OPERATIONS (WITH FLOCK) ==========

Setup: Creating test directory and file...
✓ Setup complete

TEST 1: SHARED LOCK TESTING (READ LOCKS)
----------------------------------------
  • Starting first read operation in background...
  • Starting second read operation (should succeed immediately)...
  ✓ Both read operations completed successfully (shared locking working)
  • Testing write after reads...
  ✓ Write after reads succeeded

TEST 2: EXCLUSIVE LOCK TESTING (WRITE LOCKS)
--------------------------------------------
  • Starting write operation in background (will hold exclusive lock)...
  • Attempting concurrent write operation (should block)...
  • Attempting concurrent read operation (should also block)...
  • First write operation completed, releasing lock...
  ✓ Read operation succeeded after lock release
  ✓ Second write succeeded after lock release

TEST 3: LOCK RELEASE VERIFICATION
---------------------------------
  • Verifying locks are properly released...
  ✓ Append operation succeeded
  • Reading final file contents to verify all operations worked:
  ----------------------------------------
Contents of file "lock_test_dir/locked_file.txt":
[2025-03-23 01:19:10] File created

Write after reads
First write
Second write
Final content
  ----------------------------------------

========== FILE LOCKING TESTS COMPLETED SUCCESSFULLY ==========
```

4. Conclusion

Working on this File Management System assignment was both challenging and rewarding. As a student, I found it really helpful to see how file operations work at a lower level instead of just using high-level library functions.

The biggest challenge I faced was implementing the file locking system. At first, I tried using the `fcntl()` approach with `struct flock`, which was complicated and had many fields to set correctly. I struggled with understanding all the parameters and how they worked together. After some research and experimentation, I switched to using the `flock()` system call, which was much simpler to understand and implement while still providing the same protection against multiple processes accessing files at the same time.

Error handling was also tricky. There are so many things that can go wrong when dealing with files - they might not exist, permissions might be wrong, or other processes might be using them. Making sure my program handled all these cases gracefully without crashing took a lot of testing and debugging.

I spent several hours debugging issues with the child processes created by `fork()`. Sometimes they wouldn't terminate properly or would return unexpected values. Learning how to properly manage these processes and understand how the parent and child communicate was a valuable lesson.

5. Usage

`make test` - Runs simple test mentioned in PDF.

`make comprehensive_test` - Runs custom test written by me and GPT.
