GEBZE TECHNICAL UNIVERSITY

CSE344 -- HOMEWORK #1

23/03/2025

Cemal BOLAT

STUDENT NO: 210104004010

Introduction:

This project involves the design and implementation of a Secure File and Directory Management System using the C programming language on a Linux-based system. The main objective is to provide users with a robust tool to manage files and directories securely while ensuring concurrent access is handled effectively using Linux system calls and process management techniques.

The system supports a range of fundamental file operations, including creating, listing, reading, and deleting files and directories. One key aspect of the project is the usage of process creation with fork() to handle operations concurrently. Additionally, file locking mechanisms are used to prevent concurrent write conflicts, enhancing data integrity during file modification.

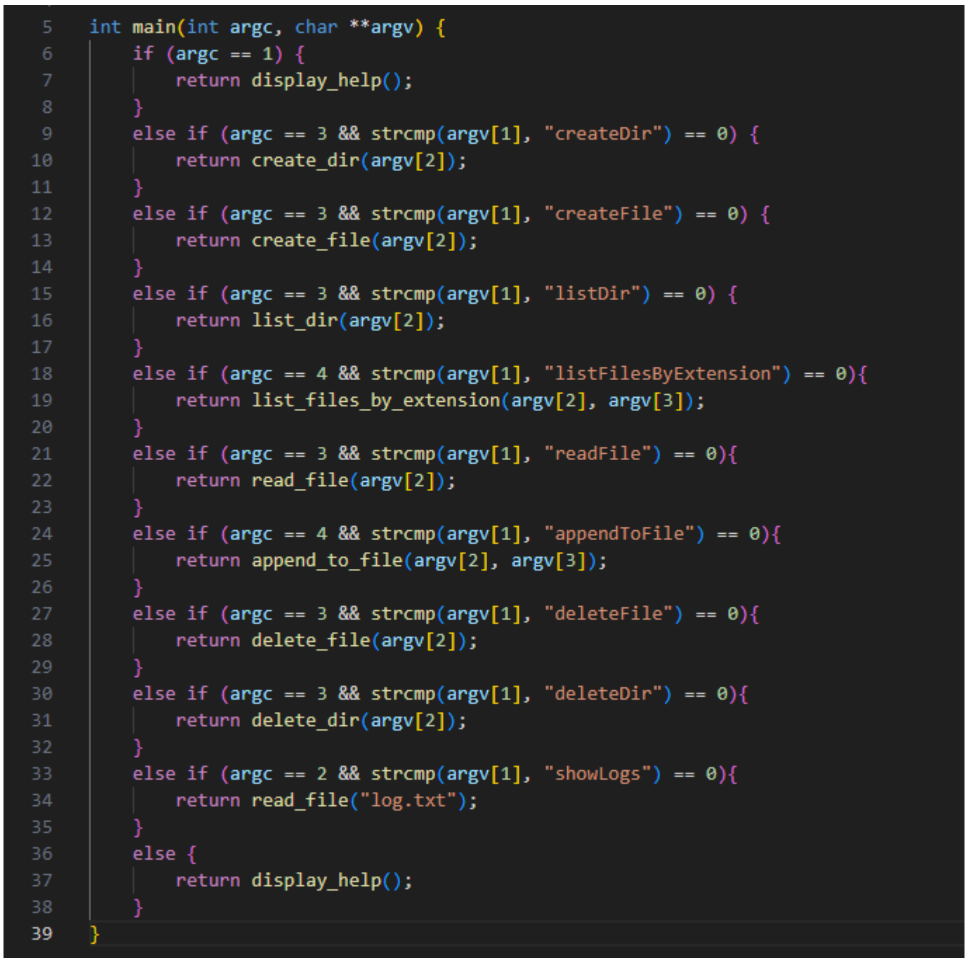
Another crucial feature of the program is the logging system, which records every action performed by the user. This allows for comprehensive auditing and monitoring of file system operations, ensuring transparency and security. The system logs are saved in a file and can be displayed to the user on demand, providing a clear view of the history of file manipulations.

Code Explanation:

main():

This function checks the command-line arguments (argc and argv) and executes different functions based on the input format.

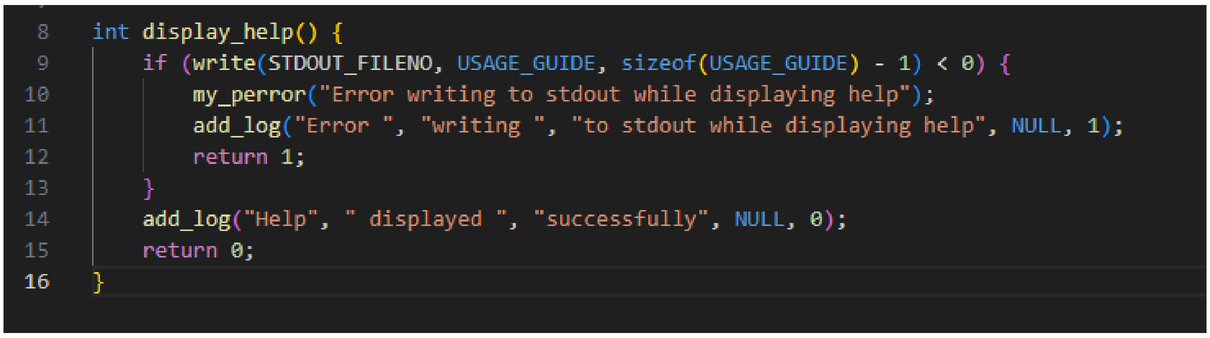
- The program processes the input arguments, checks if the required number of arguments are provided, and calls the right function accordingly.
- If there's no command or incorrect input, it shows a help message (display_help()).



```c
int main(int argc, char **argv) {
    if (argc == 1) {
        return display_help();
    }
    else if (argc == 3 && strcmp(argv[1], "createDir") == 0) {
        return create_dir(argv[2]);
    }
    else if (argc == 3 && strcmp(argv[1], "createFile") == 0) {
        return create_file(argv[2]);
    }
    else if (argc == 3 && strcmp(argv[1], "listDir") == 0) {
        return list_dir(argv[2]);
    }
    else if (argc == 4 && strcmp(argv[1], "listFilesByExtension") == 0){
        return list_files_by_extension(argv[2], argv[3]);
    }
    else if (argc == 3 && strcmp(argv[1], "readFile") == 0){
        return read_file(argv[2]);
    }
    else if (argc == 4 && strcmp(argv[1], "appendToFile") == 0){
        return append_to_file(argv[2], argv[3]);
    }
    else if (argc == 3 && strcmp(argv[1], "deleteFile") == 0){
        return delete_file(argv[2]);
    }
    else if (argc == 3 && strcmp(argv[1], "deleteDir") == 0){
        return delete_dir(argv[2]);
    }
    else if (argc == 2 && strcmp(argv[1], "showLogs") == 0){
        return read_file("log.txt");
    }
    else {
        return display_help();
    }
}
```

display_help():

This function displays a help message to the user using the write() system call. It checks if the write operation is successful. If it is, it logs the success; if there's an error while writing to the stdout, it logs the error, calls my_perror() to write error message to stderr and returns a failure code.



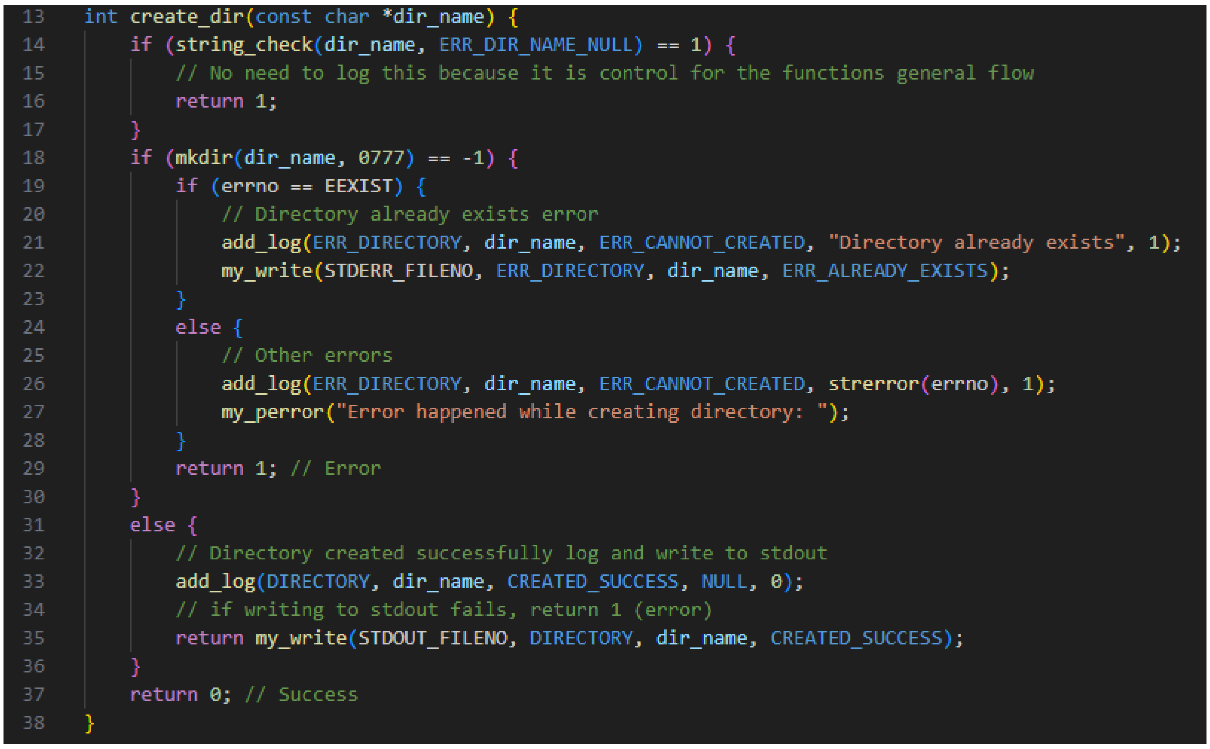
```c
int display_help() {
    if (write(STDOUT_FILENO, USAGE_GUIDE, sizeof(USAGE_GUIDE) - 1) < 0) {
        my_perror("Error writing to stdout while displaying help");
        add_log("Error ", "writing ", "to stdout while displaying help", NULL, 1);
        return 1;
    }
    add_log("Help", " displayed ", "successfully", NULL, 0);
    return 0;
}
```

create_dir():

This function attempts to create a directory with the specified name. It first checks if the directory name is valid. If it’s invalid, it returns an error code without logging, as it's just a flow control check.

If the directory name is valid, it attempts to create the directory using mkdir(). If the directory already exists, it logs the error and prints a message to stderr. If another error occurs (e.g., permission issues), it logs the error and uses my_perror() to display the error message.

If the directory is created successfully, it logs the success and writes a success message to stdout.

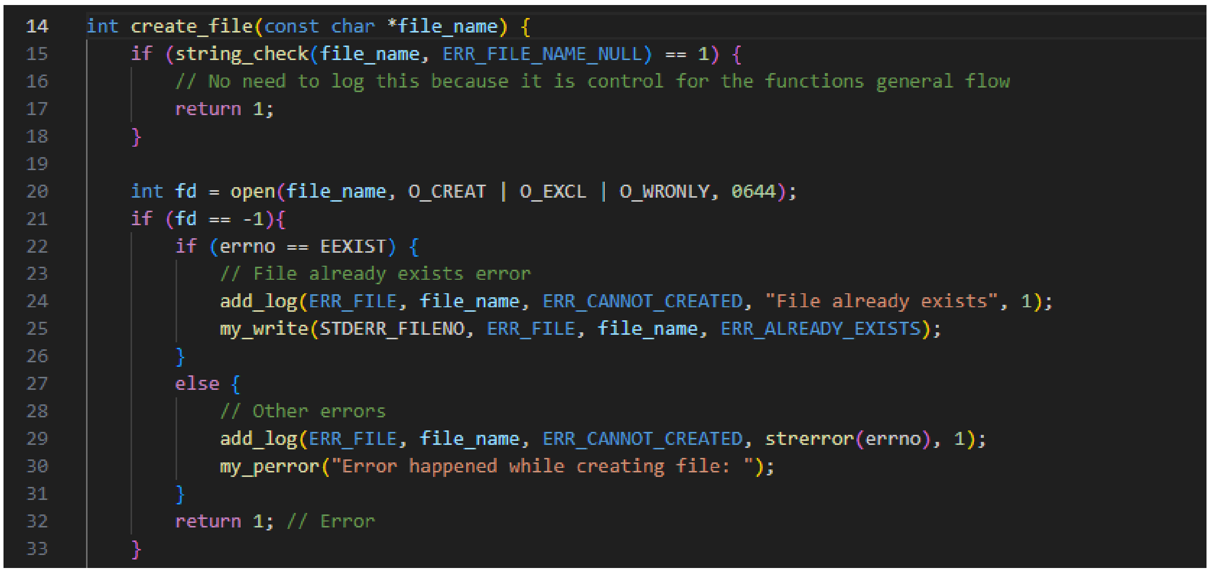


```c
int create_dir(const char *dir_name) {
    if (string_check(dir_name, ERR_DIR_NAME_NULL) == 1) {
        // No need to log this because it is control for the functions general flow
        return 1;
    }
    if (mkdir(dir_name, 0777) == -1) {
        if (errno == EEXIST) {
            // Directory already exists error
            add_log(ERR_DIRECTORY, dir_name, ERR_CANNOT_CREATED, "Directory already exists", 1);
            my_write(STDERR_FILENO, ERR_DIRECTORY, dir_name, ERR_ALREADY_EXISTS);
        }
        else {
            // Other errors
            add_log(ERR_DIRECTORY, dir_name, ERR_CANNOT_CREATED, strerror(errno), 1);
            my_perror("Error happened while creating directory: ");
        }
        return 1; // Error
    }
    else {
        // Directory created successfully log and write to stdout
        add_log(DIRECTORY, dir_name, CREATED_SUCCESS, NULL, 0);
        // if writing to stdout fails, return 1 (error)
        return my_write(STDOUT_FILENO, DIRECTORY, dir_name, CREATED_SUCCESS);
    }
    return 0; // Success
}
```

create_file():

File creation part:

The function begins by checking if the file_name is valid using string_check(). If it's invalid, the function immediately returns 1 without logging anything. Then, the function tries to create the file using the open() system call with the O_CREAT | O_EXCL | O_WRONLY flags, which ensure that the file is created only if it doesn't already exist. If the file already exists (errno == EEXIST), it logs this as an error and prints an error message to stderr. For any other error during file creation, the function logs the error, calls my_perror() to display the message, and returns 1.

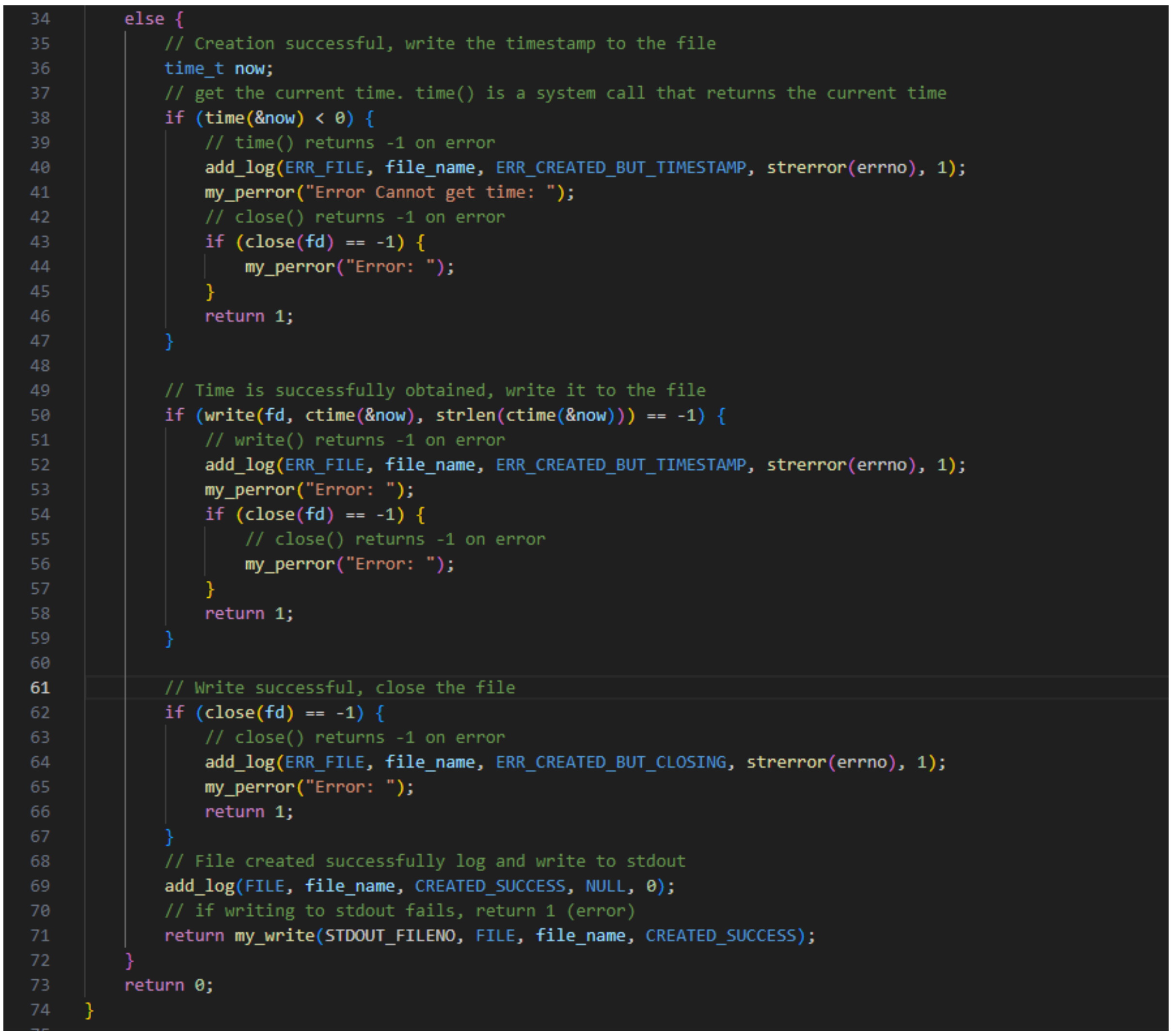


```c
int create_file(const char *file_name) {
    if (string_check(file_name, ERR_FILE_NAME_NULL) == 1) {
        // No need to log this because it is control for the functions general flow
        return 1;
    }

    int fd = open(file_name, O_CREAT | O_EXCL | O_WRONLY, 0644);
    if (fd == -1){
        if (errno == EEXIST) {
            // File already exists error
            add_log(ERR_FILE, file_name, ERR_CANNOT_CREATED, "File already exists", 1);
            my_write(STDERR_FILENO, ERR_FILE, file_name, ERR_ALREADY_EXISTS);
        }
        else {
            // Other errors
            add_log(ERR_FILE, file_name, ERR_CANNOT_CREATED, strerror(errno), 1);
            my_perror("Error happened while creating file: ");
        }
        return 1; // Error
    }
```

Writing timestamp to file is in next page.

Writing timestamp into file part:

Once the file is successfully created, the function proceeds to write the current timestamp into it. It calls the time() function to get the current time, and if this fails, it logs the error and closes the file. If the time is successfully obtained, the function writes it to the file using write(). If this write operation fails, it logs the error and closes the file as well. If everything goes smoothly, the file is closed successfully, and the function logs the success and writes a success message to stdout. If any part of this process fails, the function returns 1, indicating an error.

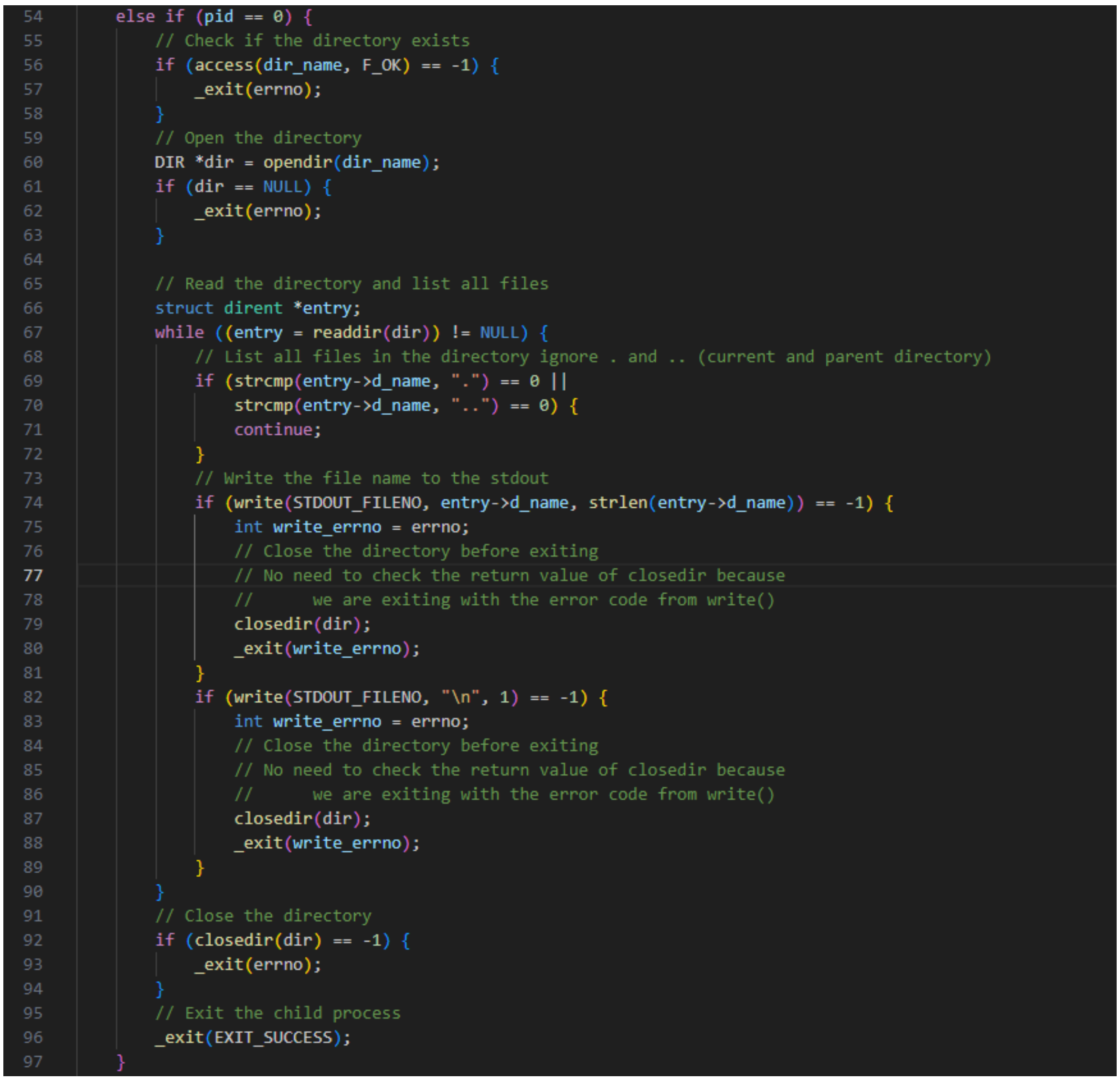


```c
    else {
        // Creation successful, write the timestamp to the file
        time_t now;
        // get the current time. time() is a system call that returns the current time
        if (time(&now) < 0) {
            // time() returns -1 on error
            add_log(ERR_FILE, file_name, ERR_CREATED_BUT_TIMESTAMP, strerror(errno), 1);
            my_perror("Error Cannot get time: ");
            // close() returns -1 on error
            if (close(fd) == -1) {
                my_perror("Error: ");
            }
            return 1;
        }

        // Time is successfully obtained, write it to the file
        if (write(fd, ctime(&now), strlen(ctime(&now))) == -1) {
            // write() returns -1 on error
            add_log(ERR_FILE, file_name, ERR_CREATED_BUT_TIMESTAMP, strerror(errno), 1);
            my_perror("Error: ");
            if (close(fd) == -1) {
                // close() returns -1 on error
                my_perror("Error: ");
            }
            return 1;
        }

        // Write successful, close the file
        if (close(fd) == -1) {
            // close() returns -1 on error
            add_log(ERR_FILE, file_name, ERR_CREATED_BUT_CLOSING, strerror(errno), 1);
            my_perror("Error: ");
            return 1;
        }
        // File created successfully log and write to stdout
        add_log(FILE, file_name, CREATED_SUCCESS, NULL, 0);
        // if writing to stdout fails, return 1 (error)
        return my_write(STDOUT_FILENO, FILE, file_name, CREATED_SUCCESS);
    }
    return 0;
}
```

list_dir():

Child process part:

In the child process (when pid == 0), the function performs the actual work of listing the directory contents. First, it checks if the directory exists by using the access() function. If the directory doesn’t exist, it exits with the corresponding error code. Then, it opens the directory using opendir(). If there’s an issue opening the directory, it exits with an error code. The child process then proceeds to read and list all files within the directory using readdir(). It ignores the entries for the current directory (".") and the parent directory (".."). Each file's name is written to stdout using the write() system call, and if an error occurs during the write operation, the directory is closed and the child process exits with the corresponding error. After successfully listing all files, the child process closes the directory and exits with a success code.



```c
    else if (pid == 0) {
        // Check if the directory exists
        if (access(dir_name, F_OK) == -1) {
            _exit(errno);
        }
        // Open the directory
        DIR *dir = opendir(dir_name);
        if (dir == NULL) {
            _exit(errno);
        }

        // Read the directory and list all files
        struct dirent *entry;
        while ((entry = readdir(dir)) != NULL) {
            // List all files in the directory ignore . and .. (current and parent directory)
            if (strcmp(entry->d_name, ".") == 0 ||
                strcmp(entry->d_name, "..") == 0) {
                continue;
            }
            // Write the file name to the stdout
            if (write(STDOUT_FILENO, entry->d_name, strlen(entry->d_name)) == -1) {
                int write_errno = errno;
                // Close the directory before exiting
                // No need to check the return value of closedir because
                //      we are exiting with the error code from write()
                closedir(dir);
                _exit(write_errno);
            }
            if (write(STDOUT_FILENO, "\n", 1) == -1) {
                int write_errno = errno;
                // Close the directory before exiting
                // No need to check the return value of closedir because
                //      we are exiting with the error code from write()
                closedir(dir);
                _exit(write_errno);
            }
        }
        // Close the directory
        if (closedir(dir) == -1) {
            _exit(errno);
        }
        // Exit the child process
        _exit(EXIT_SUCCESS);
    }
```

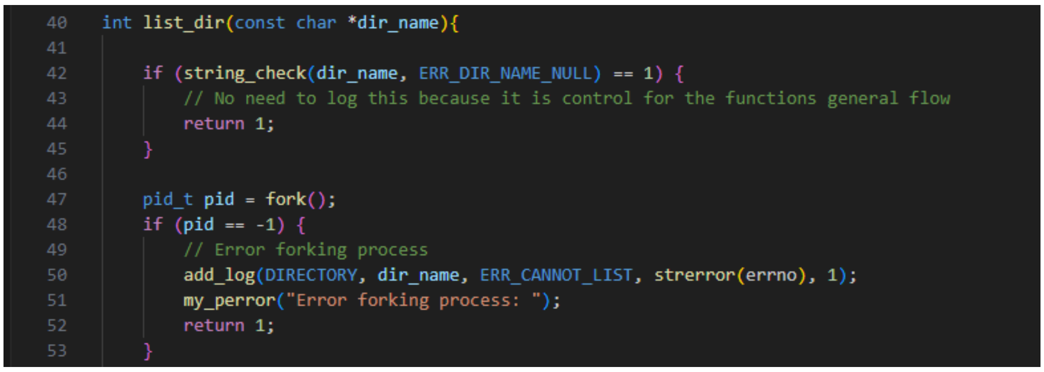
Parent process is in next page.

Parent process part:

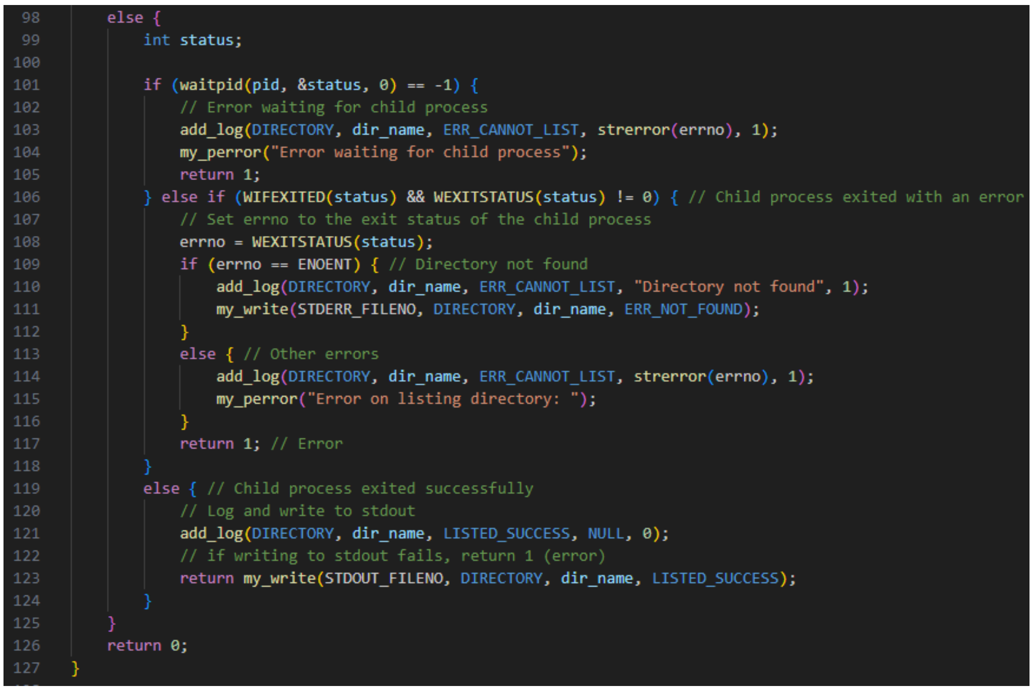
The parent process (when pid > 0) is responsible for managing the child process. After forking, it waits for the child process to complete using waitpid(). If there is an error during fork() (i.e., if pid == -1), the parent process logs an error indicating the failure to fork and prints an error message using my_perror(). The error message includes the reason for the fork() failure, and then the parent returns 1 to indicate a failure.

If waitpid() is successful, the parent checks the exit status of the child process. If the child process exits with an error (i.e., its exit status is non-zero), the parent sets errno to the child's exit status and logs the corresponding error. Specifically, if the directory was not found (ENOENT), it logs the "directory not found" error and writes an appropriate message to stderr. For any other error, it logs the error, calls my_perror(), and returns 1.

If the child process exits successfully, the parent logs the success, writes a success message to stdout, and returns a success code.



```c
int list_dir(const char *dir_name){

    if (string_check(dir_name, ERR_DIR_NAME_NULL) == 1) {
        // No need to log this because it is control for the functions general flow
        return 1;
    }

    pid_t pid = fork();
    if (pid == -1) {
        // Error forking process
        add_log(DIRECTORY, dir_name, ERR_CANNOT_LIST, strerror(errno), 1);
        my_perror("Error forking process: ");
        return 1;
    }
```

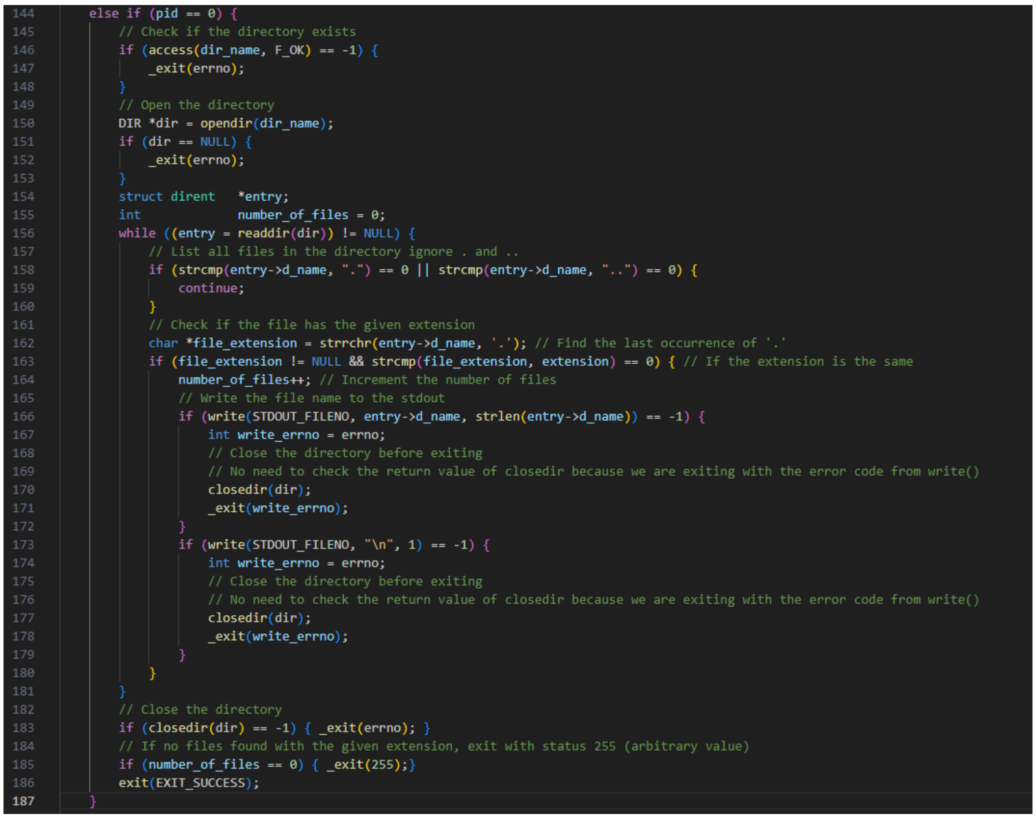


```c
    else {
        int status;

        if (waitpid(pid, &status, 0) == -1) {
            // Error waiting for child process
            add_log(DIRECTORY, dir_name, ERR_CANNOT_LIST, strerror(errno), 1);
            my_perror("Error waiting for child process");
            return 1;
        } else if (WIFEXITED(status) && WEXITSTATUS(status) != 0) { // Child process exited with an error
            // Set errno to the exit status of the child process
            errno = WEXITSTATUS(status);
            if (errno == ENOENT) { // Directory not found
                add_log(DIRECTORY, dir_name, ERR_CANNOT_LIST, "Directory not found", 1);
                my_write(STDERR_FILENO, DIRECTORY, dir_name, ERR_NOT_FOUND);
            }
            else { // Other errors
                add_log(DIRECTORY, dir_name, ERR_CANNOT_LIST, strerror(errno), 1);
                my_perror("Error on listing directory: ");
            }
            return 1; // Error
        }
        else { // Child process exited successfully
            // Log and write to stdout
            add_log(DIRECTORY, dir_name, LISTED_SUCCESS, NULL, 0);
            // if writing to stdout fails, return 1 (error)
            return my_write(STDOUT_FILENO, DIRECTORY, dir_name, LISTED_SUCCESS);
        }
    }
    return 0;
}
```

list_file_by_extension():

Child process part:

The child process is created after the fork() call, where pid == 0. Once inside the child process, the first step is to check if the specified directory exists using the access() function. If the directory does not exist, the child process immediately exits with the error code returned by access(). Next, the child opens the directory using opendir(). If the directory cannot be opened, it exits with the error code from opendir(). The child then reads the directory entries using readdir(). For each entry, it checks if the file has the specified extension using strrchr() to find the last occurrence of the dot (.) and compares it to the provided extension using strcmp(). If a file has the correct extension, it increments the count of files with that extension and writes the file name to standard output using write(). If any error occurs during writing, the child closes the directory and exits with the error code. After reading all entries in the directory, the child process closes the directory. If no files with the specified extension are found, the child exits with a status of 255 to indicate this. If everything goes smoothly, the child exits with EXIT_SUCCESS.

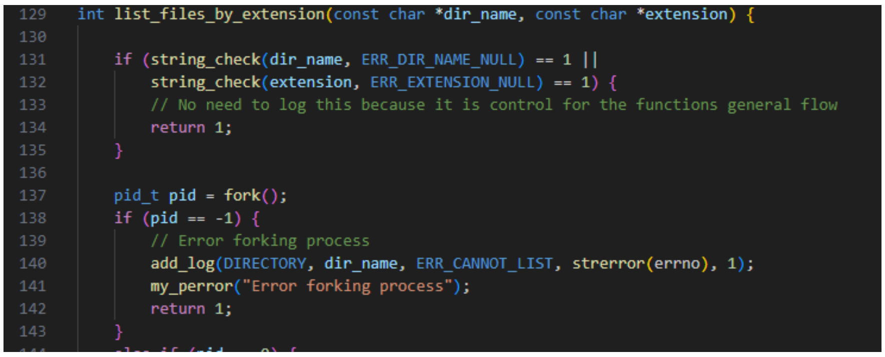


```c
else if (pid == 0) {
    // Check if the directory exists
    if (access(dir_name, F_OK) == -1) {
        _exit(errno);
    }
    // Open the directory
    DIR *dir = opendir(dir_name);
    if (dir == NULL) {
        _exit(errno);
    }
    struct dirent   *entry;
    int             number_of_files = 0;
    while ((entry = readdir(dir)) != NULL) {
        // List all files in the directory ignore . and ..
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue;
        }
        // Check if the file has the given extension
        char *file_extension = strrchr(entry->d_name, '.'); // Find the last occurrence of '.'
        if (file_extension != NULL && strcmp(file_extension, extension) == 0) { // If the extension is the same
            number_of_files++; // Increment the number of files
            // Write the file name to the stdout
            if (write(STDOUT_FILENO, entry->d_name, strlen(entry->d_name)) == -1) {
                int write_errno = errno;
                // Close the directory before exiting
                // No need to check the return value of closedir because we are exiting with the error code from write()
                closedir(dir);
                _exit(write_errno);
            }
            if (write(STDOUT_FILENO, "\n", 1) == -1) {
                int write_errno = errno;
                // Close the directory before exiting
                // No need to check the return value of closedir because we are exiting with the error code from write()
                closedir(dir);
                _exit(write_errno);
            }
        }
    }
    // Close the directory
    if (closedir(dir) == -1) { _exit(errno); }
    // If no files found with the given extension, exit with status 255 (arbitrary value)
    if (number_of_files == 0) { _exit(255);}
    exit(EXIT_SUCCESS);
}
```

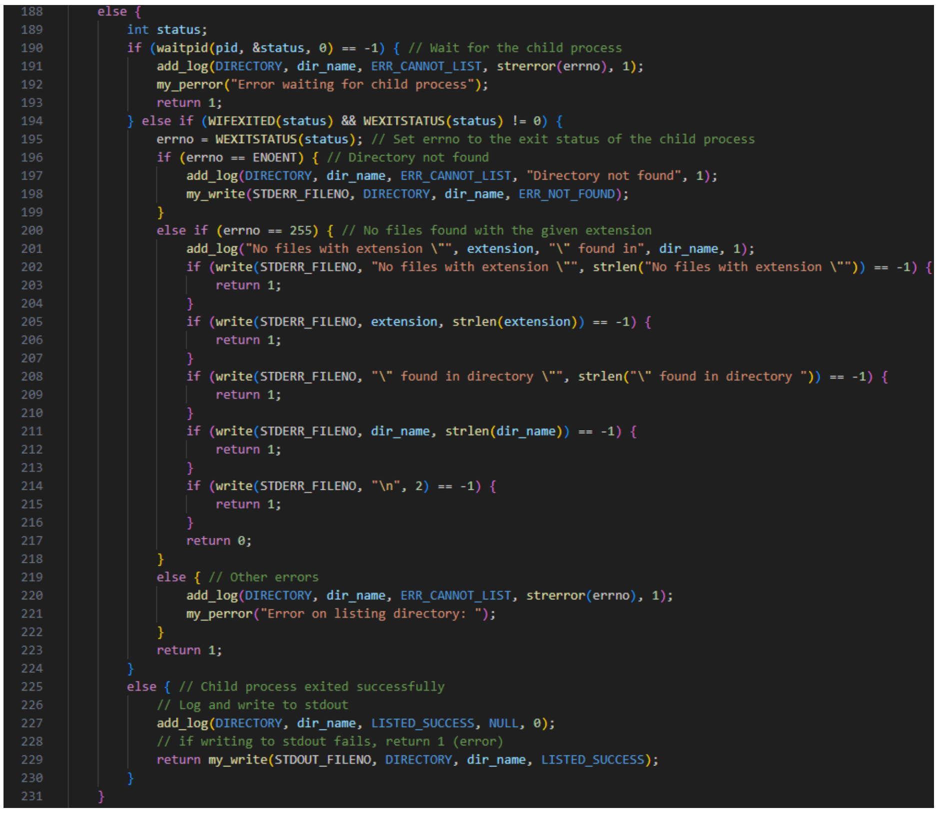
Parent process is in next page.

Parent process part:

In the parent process, after forking, the parent waits for the child process to finish using waitpid(). If there is an error in waiting for the child process, the parent logs the error and prints an error message using my_perror(), then returns 1. If the child process exits with an error code (non-zero status), the parent sets errno to the child's exit status. If the directory was not found (indicated by ENOENT), the parent logs an appropriate error message and writes it to standard error. If the child exits with status 255 (meaning no files with the specified extension were found), the parent writes a message to stderr saying no such files were found. For any other exit status from the child, the parent logs the error, prints the error message using my_perror(), and returns 1. If the child exits successfully, the parent logs the success and writes a success message to stdout. The function then returns 0, signaling that everything completed successfully.



```c
int list_files_by_extension(const char *dir_name, const char *extension) {

    if (string_check(dir_name, ERR_DIR_NAME_NULL) == 1 ||
        string_check(extension, ERR_EXTENSION_NULL) == 1) {
        // No need to log this because it is control for the functions general flow
        return 1;
    }

    pid_t pid = fork();
    if (pid == -1) {
        // Error forking process
        add_log(DIRECTORY, dir_name, ERR_CANNOT_LIST, strerror(errno), 1);
        my_perror("Error forking process");
        return 1;
    }
```



```c
    else {
        int status;
        if (waitpid(pid, &status, 0) == -1) { // Wait for the child process
            add_log(DIRECTORY, dir_name, ERR_CANNOT_LIST, strerror(errno), 1);
            my_perror("Error waiting for child process");
            return 1;
        } else if (WIFEXITED(status) && WEXITSTATUS(status) != 0) {
            errno = WEXITSTATUS(status); // Set errno to the exit status of the child process
            if (errno == ENOENT) { // Directory not found
                add_log(DIRECTORY, dir_name, ERR_CANNOT_LIST, "Directory not found", 1);
                my_write(STDERR_FILENO, DIRECTORY, dir_name, ERR_NOT_FOUND);
            }
            else if (errno == 255) { // No files found with the given extension
                add_log("No files with extension \"", extension, "\" found in", dir_name, 1);
                if (write(STDERR_FILENO, "No files with extension \"", strlen("No files with extension \"")) == -1) {
                    return 1;
                }
                if (write(STDERR_FILENO, extension, strlen(extension)) == -1) {
                    return 1;
                }
                if (write(STDERR_FILENO, "\" found in directory \"", strlen("\" found in directory ")) == -1) {
                    return 1;
                }
                if (write(STDERR_FILENO, dir_name, strlen(dir_name)) == -1) {
                    return 1;
                }
                if (write(STDERR_FILENO, "\n", 2) == -1) {
                    return 1;
                }
                return 0;
            }
            else { // Other errors
                add_log(DIRECTORY, dir_name, ERR_CANNOT_LIST, strerror(errno), 1);
                my_perror("Error on listing directory: ");
            }
            return 1;
        }
        else { // Child process exited successfully
            // Log and write to stdout
            add_log(DIRECTORY, dir_name, LISTED_SUCCESS, NULL, 0);
            // if writing to stdout fails, return 1 (error)
            return my_write(STDOUT_FILENO, DIRECTORY, dir_name, LISTED_SUCCESS);
        }
    }
```

read_file():

The read_file function is responsible for opening a file, reading its contents byte by byte, and writing them to standard output (stdout).

Initially, the function checks if the file_name is valid using string_check(). If the filename is invalid, the function simply returns 1 to indicate an error.

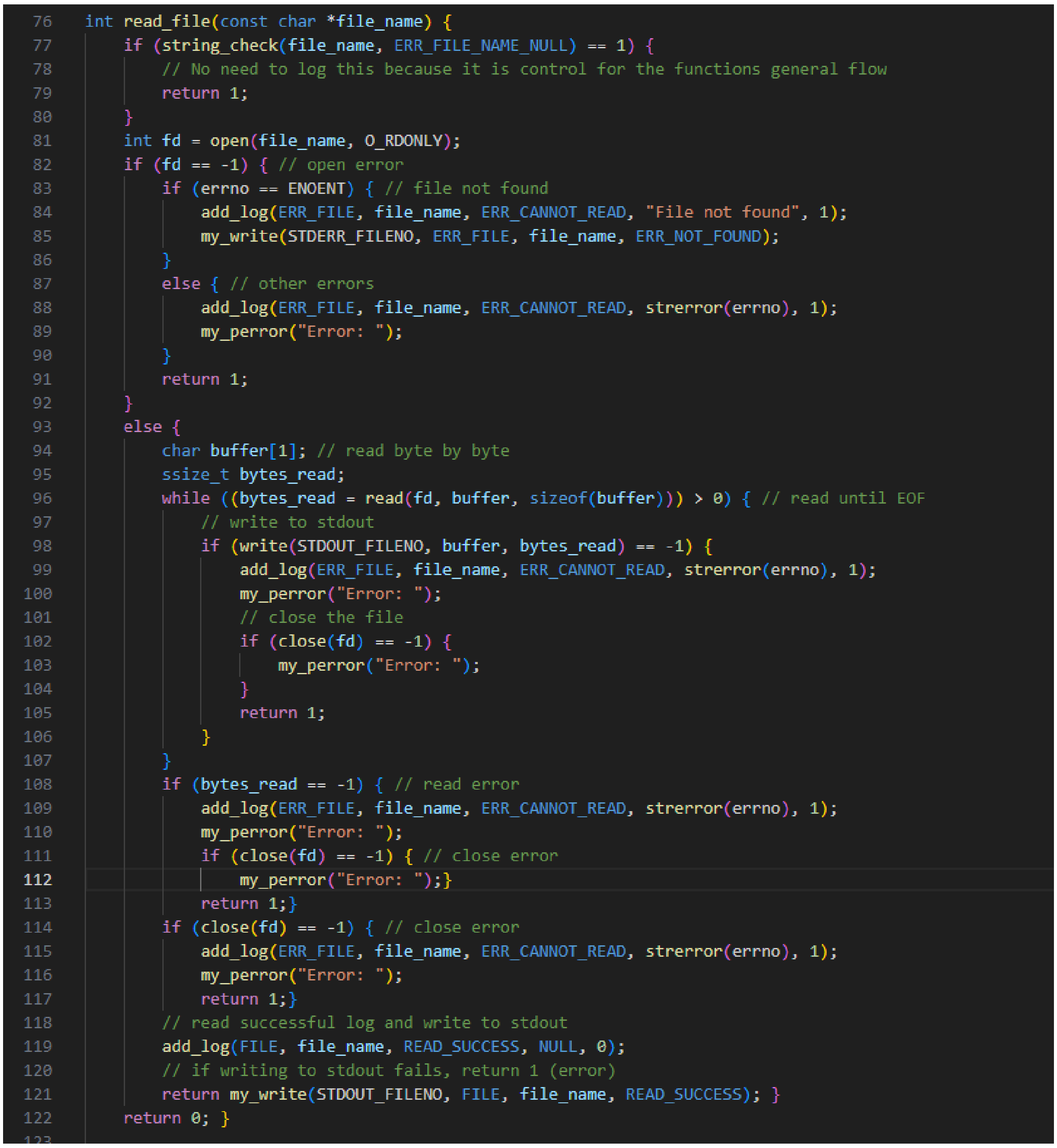
If the filename is valid, the function proceeds by attempting to open the file in read-only mode using open(). If the file cannot be opened (i.e., open() returns -1), the function checks the value of errno. If the error is ENOENT (file not found), it logs this error and writes a corresponding message to stderr. For other types of errors, it logs the error message and calls my_perror() to print the error.

If the file is successfully opened, the function enters a loop where it reads the file content byte by byte using read(). After each byte is read, it is written to stdout using the write() function. If writing to stdout fails, the function logs the error, calls my_perror(), and ensures the file is closed before exiting with an error.

Once the file is completely read, the function checks for any read errors (i.e., if bytes_read == -1). If there was an error during the read operation, it logs the error, attempts to close the file, and then returns 1 to indicate failure.

After successfully reading the entire file, the function proceeds to close the file using close(). If there is an error while closing the file, it logs the error and calls my_perror() before returning 1. If all operations are successful, the function logs the success of the file reading operation and writes a success message to stdout. Finally, it returns 0 to indicate that the file was read successfully.

Function photo is in next page.



```c
int read_file(const char *file_name) {
    if (string_check(file_name, ERR_FILE_NAME_NULL) == 1) {
        // No need to log this because it is control for the functions general flow
        return 1;
    }
    int fd = open(file_name, O_RDONLY);
    if (fd == -1) { // open error
        if (errno == ENOENT) { // file not found
            add_log(ERR_FILE, file_name, ERR_CANNOT_READ, "File not found", 1);
            my_write(STDERR_FILENO, ERR_FILE, file_name, ERR_NOT_FOUND);
        }
        else { // other errors
            add_log(ERR_FILE, file_name, ERR_CANNOT_READ, strerror(errno), 1);
            my_perror("Error: ");
        }
        return 1;
    }
    else {
        char buffer[1]; // read byte by byte
        ssize_t bytes_read;
        while ((bytes_read = read(fd, buffer, sizeof(buffer))) > 0) { // read until EOF
            // write to stdout
            if (write(STDOUT_FILENO, buffer, bytes_read) == -1) {
                add_log(ERR_FILE, file_name, ERR_CANNOT_READ, strerror(errno), 1);
                my_perror("Error: ");
                // close the file
                if (close(fd) == -1) {
                    my_perror("Error: ");
                }
                return 1;
            }
        }
        if (bytes_read == -1) { // read error
            add_log(ERR_FILE, file_name, ERR_CANNOT_READ, strerror(errno), 1);
            my_perror("Error: ");
            if (close(fd) == -1) { // close error
                my_perror("Error: ");}
            return 1;}
        if (close(fd) == -1) { // close error
            add_log(ERR_FILE, file_name, ERR_CANNOT_READ, strerror(errno), 1);
            my_perror("Error: ");
            return 1;}
        // read successful log and write to stdout
        add_log(FILE, file_name, READ_SUCCESS, NULL, 0);
        // if writing to stdout fails, return 1 (error)
        return my_write(STDOUT_FILENO, FILE, file_name, READ_SUCCESS); }
    return 0; }
```

append_to_file():

The append_to_file function is responsible for appending content to a specified file. The process starts by validating the inputs: if either the file_name or content is invalid (i.e., NULL or empty), the function returns 1 without performing any file operations, as it is a control check for the function's general flow.

If the inputs are valid, the function attempts to open the specified file in append mode (O_WRONLY | O_APPEND). If the file cannot be opened (i.e., open() returns -1), the function checks the value of errno. If the error is ENOENT (file not found), it logs the error and writes a corresponding message to stderr. If the error is EACCES (permission denied), it logs a specific error indicating that the file is locked and writes this to stderr. For other errors, it logs the error message and calls my_perror() to print the error to stderr.

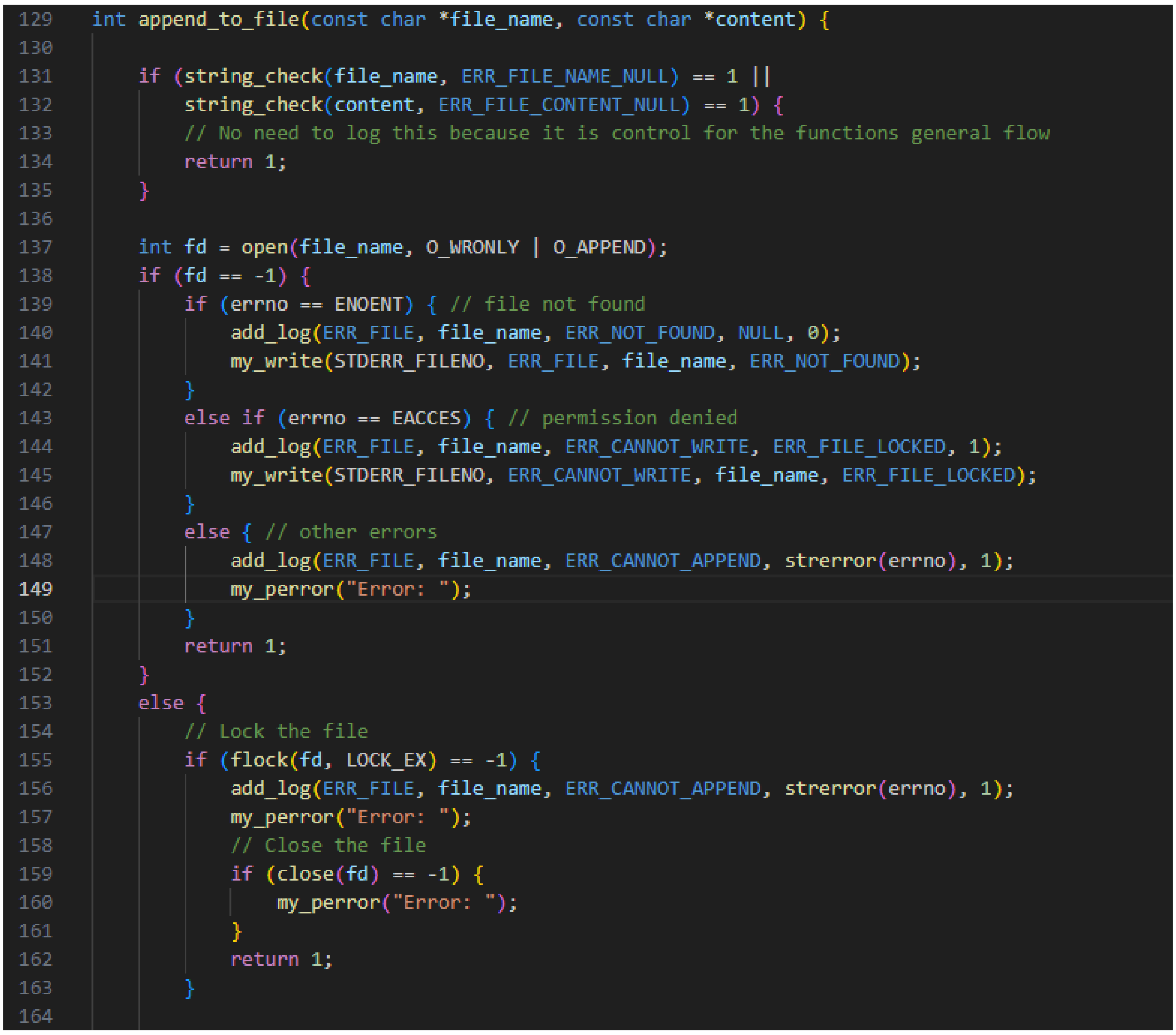
If the file is successfully opened, the function proceeds to lock the file using flock() with LOCK_EX to ensure exclusive access to the file. If the file cannot be locked (i.e., flock() fails), it logs the error, attempts to close the file, and returns 1 to indicate failure.

After the file is successfully locked, the function writes the content to the file using the write() function. If writing to the file fails (i.e., write() returns -1), the function logs the error, unlocks the file with flock() (if possible), closes the file, and returns 1.

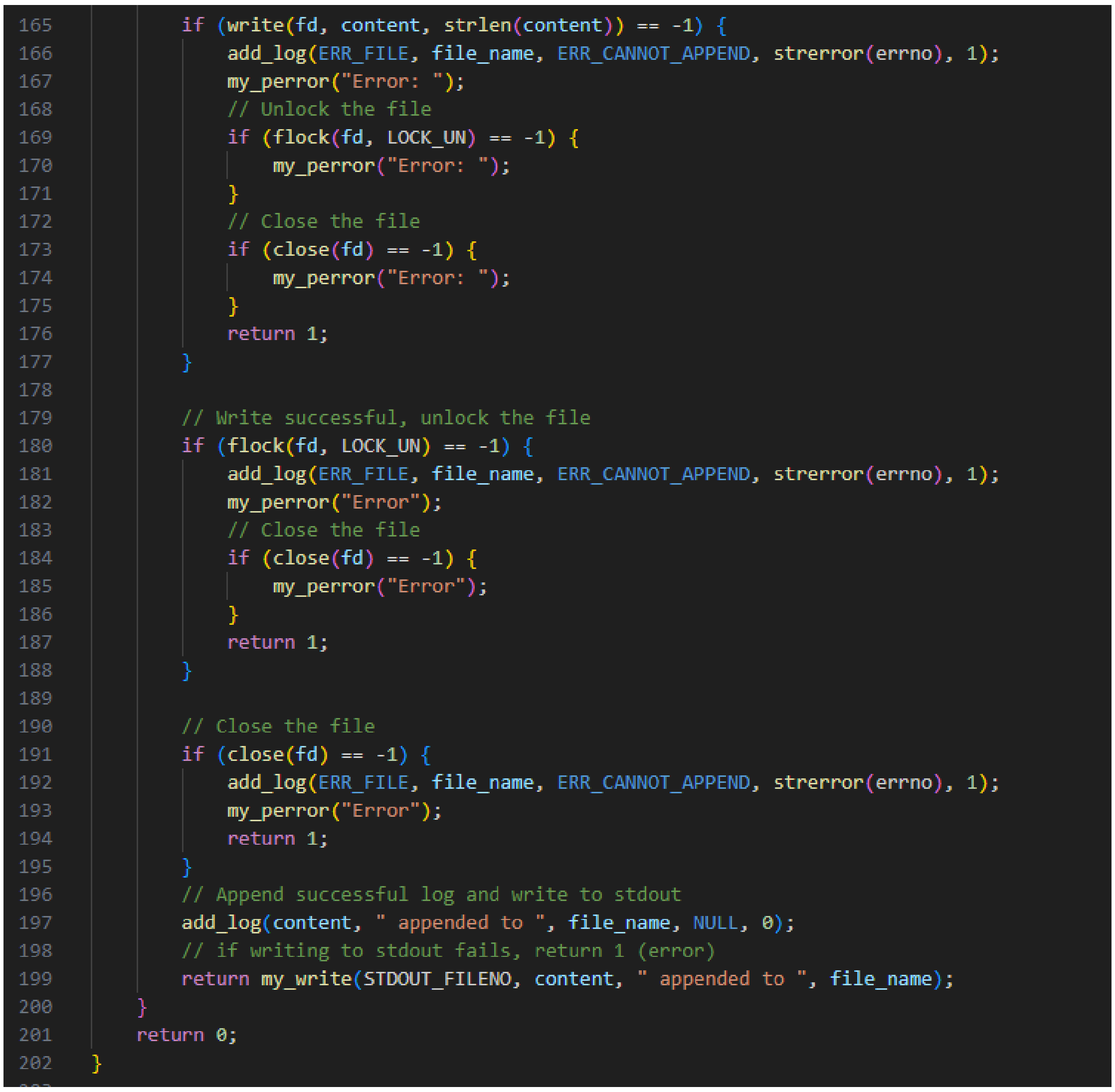
If the content is successfully written to the file, the function unlocks the file using flock() with LOCK_UN. If unlocking the file fails, it logs the error, closes the file, and returns 1. If all operations are successful, the function then proceeds to close the file. If the file cannot be closed, it logs the error and returns 1.

Finally, after all operations are successfully completed, the function logs the success of appending the content to the file and writes a success message to stdout. It returns 0 to indicate that the append operation was successful.

Function photo is in next page



```c
int append_to_file(const char *file_name, const char *content) {

    if (string_check(file_name, ERR_FILE_NAME_NULL) == 1 ||
        string_check(content, ERR_FILE_CONTENT_NULL) == 1) {
        // No need to log this because it is control for the functions general flow
        return 1;
    }

    int fd = open(file_name, O_WRONLY | O_APPEND);
    if (fd == -1) {
        if (errno == ENOENT) { // file not found
            add_log(ERR_FILE, file_name, ERR_NOT_FOUND, NULL, 0);
            my_write(STDERR_FILENO, ERR_FILE, file_name, ERR_NOT_FOUND);
        }
        else if (errno == EACCES) { // permission denied
            add_log(ERR_FILE, file_name, ERR_CANNOT_WRITE, ERR_FILE_LOCKED, 1);
            my_write(STDERR_FILENO, ERR_CANNOT_WRITE, file_name, ERR_FILE_LOCKED);
        }
        else { // other errors
            add_log(ERR_FILE, file_name, ERR_CANNOT_APPEND, strerror(errno), 1);
            my_perror("Error: ");
        }
        return 1;
    }
    else {
        // Lock the file
        if (flock(fd, LOCK_EX) == -1) {
            add_log(ERR_FILE, file_name, ERR_CANNOT_APPEND, strerror(errno), 1);
            my_perror("Error: ");
            // Close the file
            if (close(fd) == -1) {
                my_perror("Error: ");
            }
            return 1;
        }

```



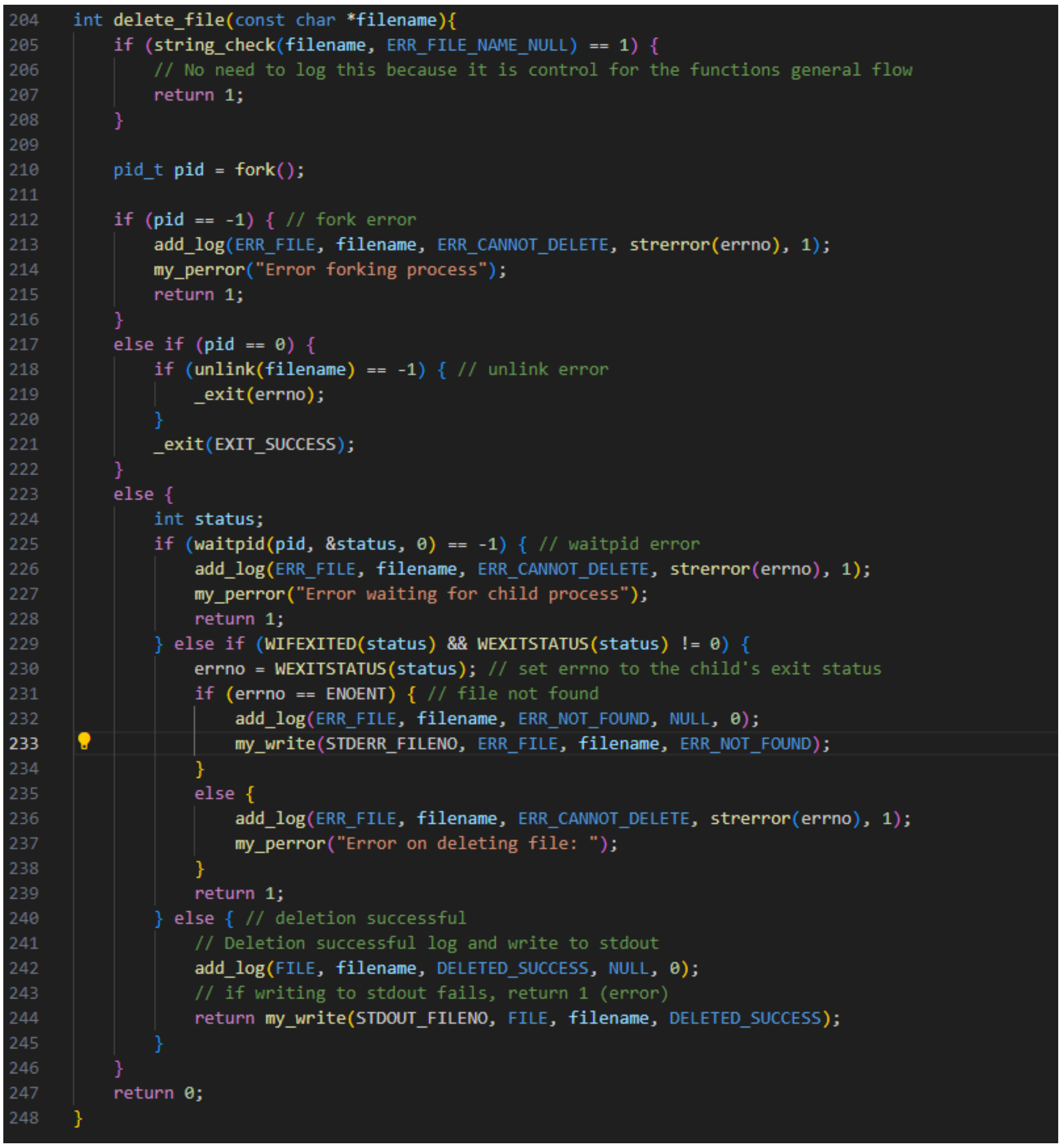
```c
        if (write(fd, content, strlen(content)) == -1) {
            add_log(ERR_FILE, file_name, ERR_CANNOT_APPEND, strerror(errno), 1);
            my_perror("Error: ");
            // Unlock the file
            if (flock(fd, LOCK_UN) == -1) {
                my_perror("Error: ");
            }
            // Close the file
            if (close(fd) == -1) {
                my_perror("Error: ");
            }
            return 1;
        }

        // Write successful, unlock the file
        if (flock(fd, LOCK_UN) == -1) {
            add_log(ERR_FILE, file_name, ERR_CANNOT_APPEND, strerror(errno), 1);
            my_perror("Error");
            // Close the file
            if (close(fd) == -1) {
                my_perror("Error");
            }
            return 1;
        }

        // Close the file
        if (close(fd) == -1) {
            add_log(ERR_FILE, file_name, ERR_CANNOT_APPEND, strerror(errno), 1);
            my_perror("Error");
            return 1;
        }
        // Append successful log and write to stdout
        add_log(content, " appended to ", file_name, NULL, 0);
        // if writing to stdout fails, return 1 (error)
        return my_write(STDOUT_FILENO, content, " appended to ", file_name);
    }
    return 0;
}
```

delete_file():

The delete_file function attempts to delete a specified file. It first checks if the filename is valid and not NULL. If it's invalid, the function returns 1.

It then creates a child process using fork(). If fork() fails, it logs the error and returns 1. In the child process, it tries to delete the file with unlink(). If the deletion fails, it exits with the error code. If successful, it exits with EXIT_SUCCESS.

The parent process waits for the child to finish. If waitpid() fails, it logs the error and returns 1. If the child exits with an error, the parent logs the error (e.g., file not found) and returns 1. If successful, the parent logs the success, writes a success message to stdout, and returns 0.



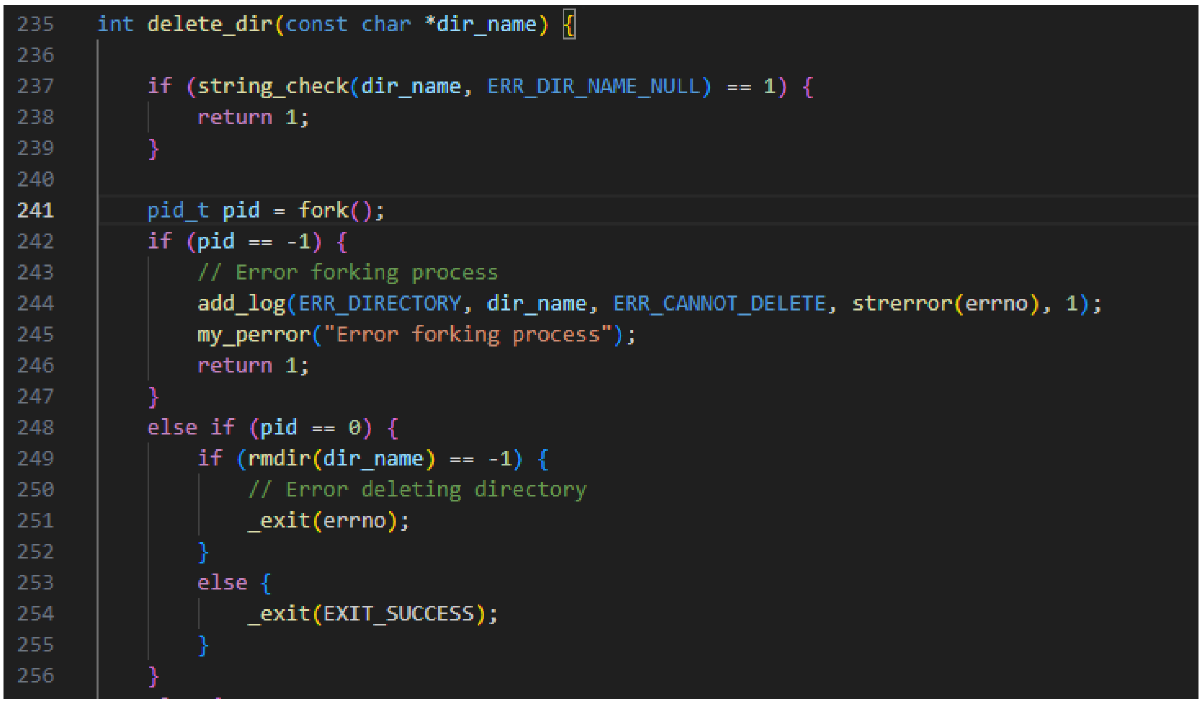
```c
int delete_file(const char *filename){
    if (string_check(filename, ERR_FILE_NAME_NULL) == 1) {
        // No need to log this because it is control for the functions general flow
        return 1;
    }

    pid_t pid = fork();

    if (pid == -1) { // fork error
        add_log(ERR_FILE, filename, ERR_CANNOT_DELETE, strerror(errno), 1);
        my_perror("Error forking process");
        return 1;
    }
    else if (pid == 0) {
        if (unlink(filename) == -1) { // unlink error
            _exit(errno);
        }
        _exit(EXIT_SUCCESS);
    }
    else {
        int status;
        if (waitpid(pid, &status, 0) == -1) { // waitpid error
            add_log(ERR_FILE, filename, ERR_CANNOT_DELETE, strerror(errno), 1);
            my_perror("Error waiting for child process");
            return 1;
        } else if (WIFEXITED(status) && WEXITSTATUS(status) != 0) {
            errno = WEXITSTATUS(status); // set errno to the child's exit status
            if (errno == ENOENT) { // file not found
                add_log(ERR_FILE, filename, ERR_NOT_FOUND, NULL, 0);
                my_write(STDERR_FILENO, ERR_FILE, filename, ERR_NOT_FOUND);
            }
            else {
                add_log(ERR_FILE, filename, ERR_CANNOT_DELETE, strerror(errno), 1);
                my_perror("Error on deleting file: ");
            }
            return 1;
        } else { // deletion successful
            // Deletion successful log and write to stdout
            add_log(FILE, filename, DELETED_SUCCESS, NULL, 0);
            // if writing to stdout fails, return 1 (error)
            return my_write(STDOUT_FILENO, FILE, filename, DELETED_SUCCESS);
        }
    }
    return 0;
}
```

delete_directory():

The delete_dir function attempts to delete a specified directory. First, it checks if the directory name (dir_name) is valid and not NULL. If it's invalid, the function returns 1.

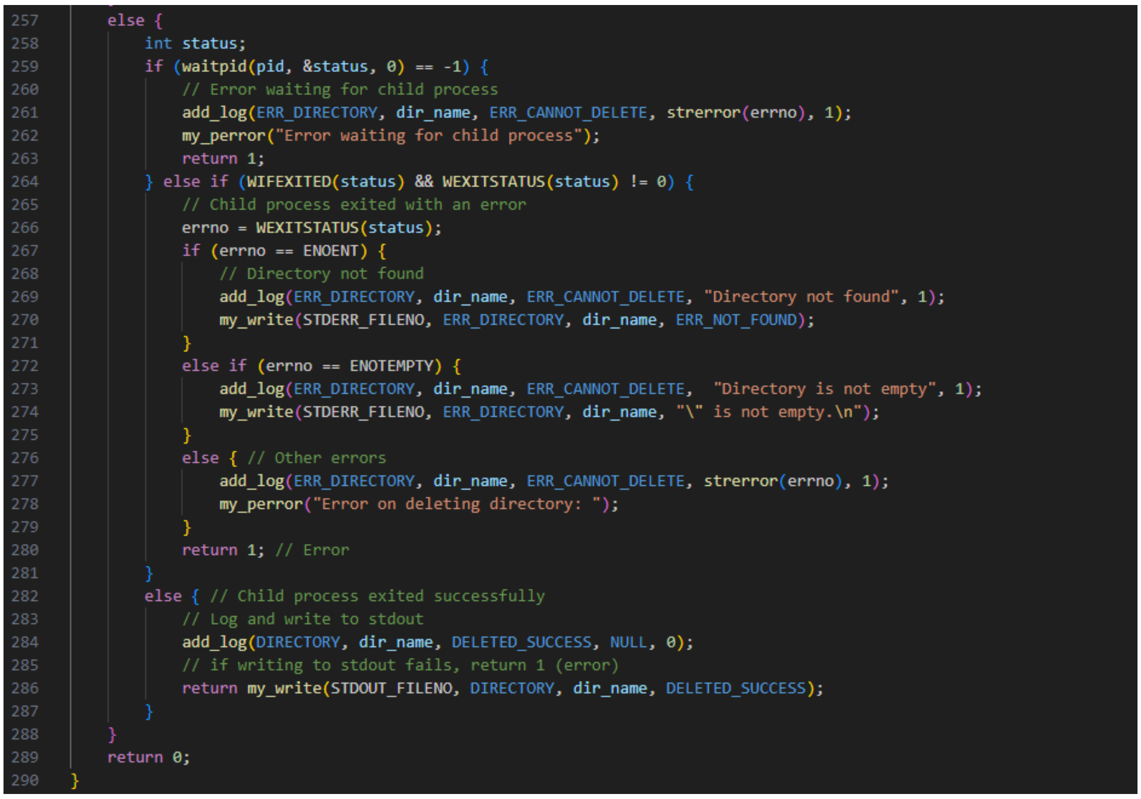
The function then creates a child process using fork(). If the fork() fails, it logs the error and returns 1. In the child process, it attempts to delete the directory using rmdir(). If the deletion fails, it exits with the error code; otherwise, it exits successfully.

In the parent process, it waits for the child process to complete with waitpid(). If waiting for the child fails, it logs the error and returns 1. If the child process exits with an error (e.g., directory not found, directory not empty), it logs the corresponding error and returns 1. If the deletion is successful, it logs the success and writes a success message to stdout, then returns 0.



```c
int delete_dir(const char *dir_name) {

    if (string_check(dir_name, ERR_DIR_NAME_NULL) == 1) {
        return 1;
    }

    pid_t pid = fork();
    if (pid == -1) {
        // Error forking process
        add_log(ERR_DIRECTORY, dir_name, ERR_CANNOT_DELETE, strerror(errno), 1);
        my_perror("Error forking process");
        return 1;
    }
    else if (pid == 0) {
        if (rmdir(dir_name) == -1) {
            // Error deleting directory
            _exit(errno);
        }
        else {
            _exit(EXIT_SUCCESS);
        }
    }
```

Parent process part is in next page



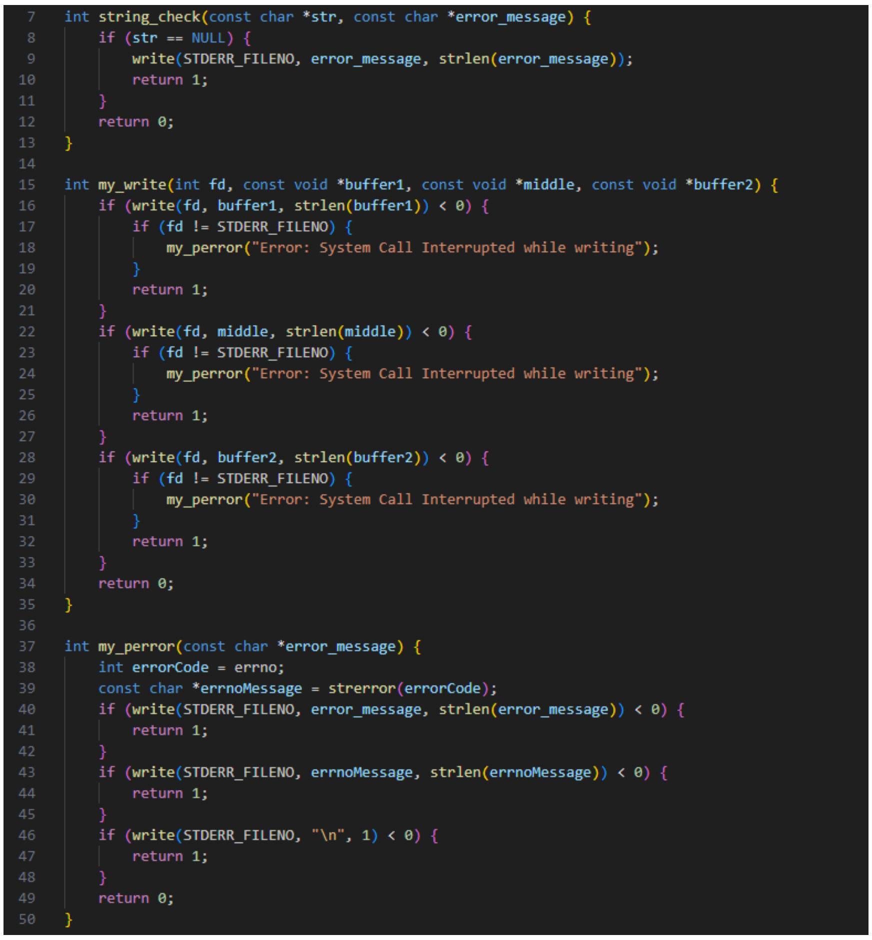
```c
    else {
        int status;
        if (waitpid(pid, &status, 0) == -1) {
            // Error waiting for child process
            add_log(ERR_DIRECTORY, dir_name, ERR_CANNOT_DELETE, strerror(errno), 1);
            my_perror("Error waiting for child process");
            return 1;
        } else if (WIFEXITED(status) && WEXITSTATUS(status) != 0) {
            // Child process exited with an error
            errno = WEXITSTATUS(status);
            if (errno == ENOENT) {
                // Directory not found
                add_log(ERR_DIRECTORY, dir_name, ERR_CANNOT_DELETE, "Directory not found", 1);
                my_write(STDERR_FILENO, ERR_DIRECTORY, dir_name, ERR_NOT_FOUND);
            }
            else if (errno == ENOTEMPTY) {
                add_log(ERR_DIRECTORY, dir_name, ERR_CANNOT_DELETE,  "Directory is not empty", 1);
                my_write(STDERR_FILENO, ERR_DIRECTORY, dir_name, "\" is not empty.\n");
            }
            else { // Other errors
                add_log(ERR_DIRECTORY, dir_name, ERR_CANNOT_DELETE, strerror(errno), 1);
                my_perror("Error on deleting directory: ");
            }
            return 1; // Error
        }
        else { // Child process exited successfully
            // Log and write to stdout
            add_log(DIRECTORY, dir_name, DELETED_SUCCESS, NULL, 0);
            // if writing to stdout fails, return 1 (error)
            return my_write(STDOUT_FILENO, DIRECTORY, dir_name, DELETED_SUCCESS);
        }
    }
    return 0;
}
```

UTILS FUNCTIONS:

string_check: This function checks if the provided string (str) is NULL. If the string is NULL, it writes the provided error_message to stderr and returns 1 to indicate an error. it simply returns 0, indicating no issue with the string.

my_write: This function is responsible for writing three separate pieces of data (buffer1, middle, buffer2) to the specified file descriptor (fd). It first attempts to write buffer1, then middle, and finally buffer2. If any of the write() operations fail (i.e., return a value less than 0), it logs an error message (if the file descriptor is not STDERR_FILENO) and returns 1.

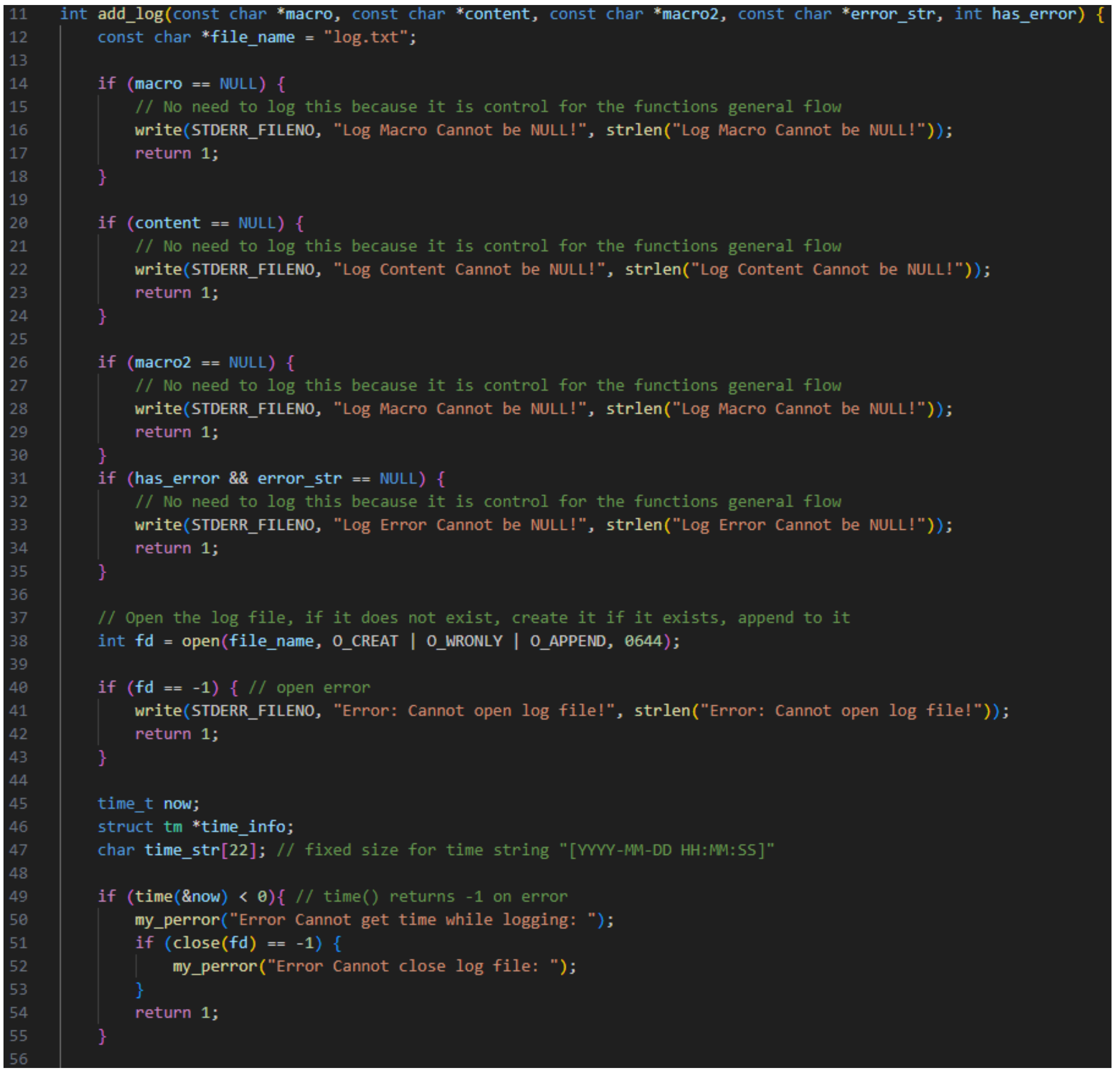
my_perror: This function is used to print an error message along with the system error message that corresponds to the last occurred error. It takes an error_message as an argument and first writes this custom message to stderr. Then, it retrieves the system error message associated with the last error (using errno) and writes it to stderr as well. Finally, it appends a newline. If any of the write() operations fail while printing, it returns 1 to indicate failure. Otherwise, it returns 0, meaning the error message was successfully printed.



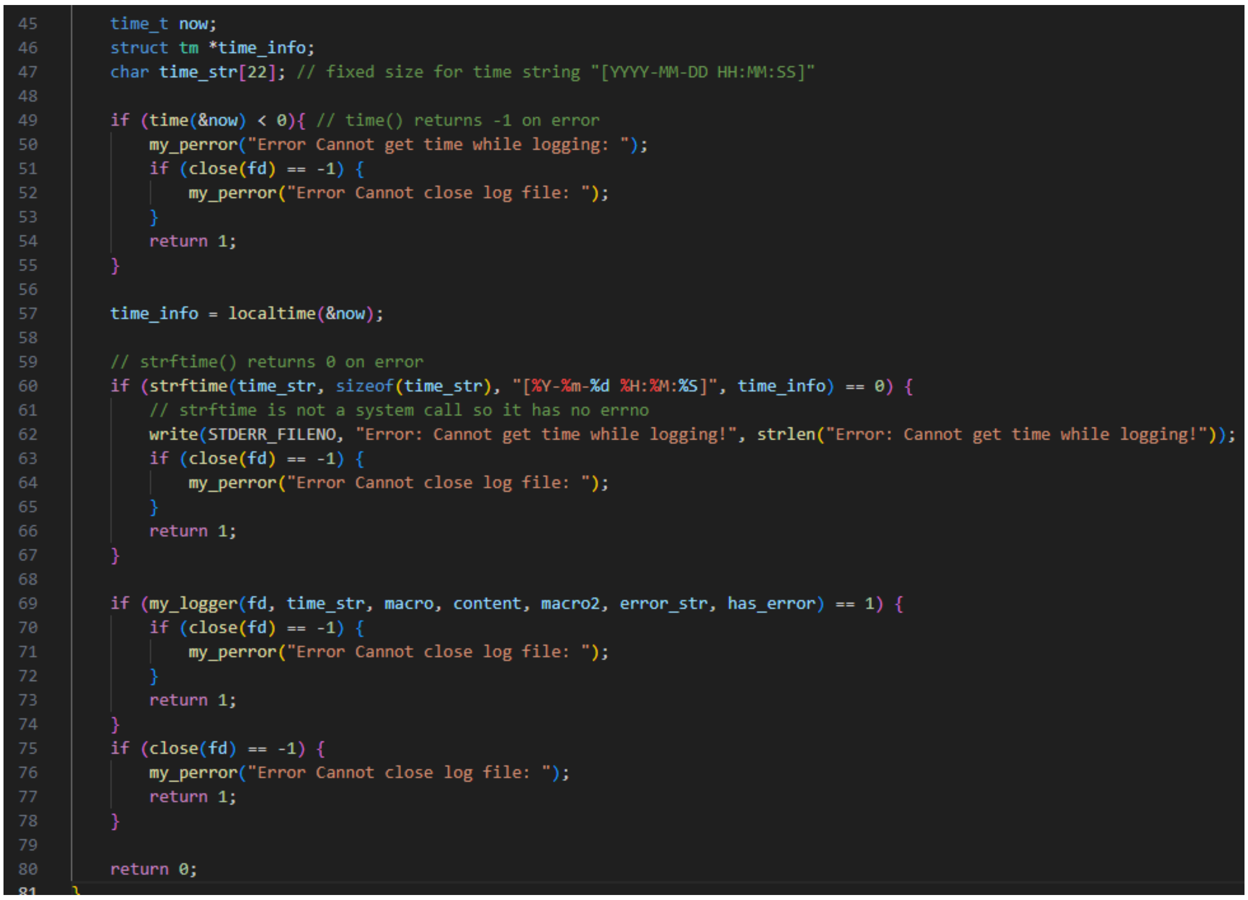
```c
int string_check(const char *str, const char *error_message) {
    if (str == NULL) {
        write(STDERR_FILENO, error_message, strlen(error_message));
        return 1;
    }
    return 0;
}

int my_write(int fd, const void *buffer1, const void *middle, const void *buffer2) {
    if (write(fd, buffer1, strlen(buffer1)) < 0) {
        if (fd != STDERR_FILENO) {
            my_perror("Error: System Call Interrupted while writing");
        }
        return 1;
    }
    if (write(fd, middle, strlen(middle)) < 0) {
        if (fd != STDERR_FILENO) {
            my_perror("Error: System Call Interrupted while writing");
        }
        return 1;
    }
    if (write(fd, buffer2, strlen(buffer2)) < 0) {
        if (fd != STDERR_FILENO) {
            my_perror("Error: System Call Interrupted while writing");
        }
        return 1;
    }
    return 0;
}

int my_perror(const char *error_message) {
    int errorCode = errno;
    const char *errnoMessage = strerror(errorCode);
    if (write(STDERR_FILENO, error_message, strlen(error_message)) < 0) {
        return 1;
    }
    if (write(STDERR_FILENO, errnoMessage, strlen(errnoMessage)) < 0) {
        return 1;
    }
    if (write(STDERR_FILENO, "\n", 1) < 0) {
        return 1;
    }
    return 0;
}
```

LOG FUNCTIONS

add_log function is designed to log events into a file named log.txt. It accepts several parameters: macro, content, macro2, and error_str, which make up the log message, and a flag has_error indicating whether to include an error message. The function first checks if any of the required parameters are NULL and writes an error message to stderr and returns 1 if any are found. It proceeds by opening the log.txt file in append mode, creating the file if it doesn't exist, and handling any errors that may occur during this process. If the file is successfully opened, the current timestamp is retrieved and formatted into a string. Then, it calls the my_logger function to write the actual log entry to the file. Finally, the log file is closed, and the function returns 0 on success or 1 if an error occurs at any point

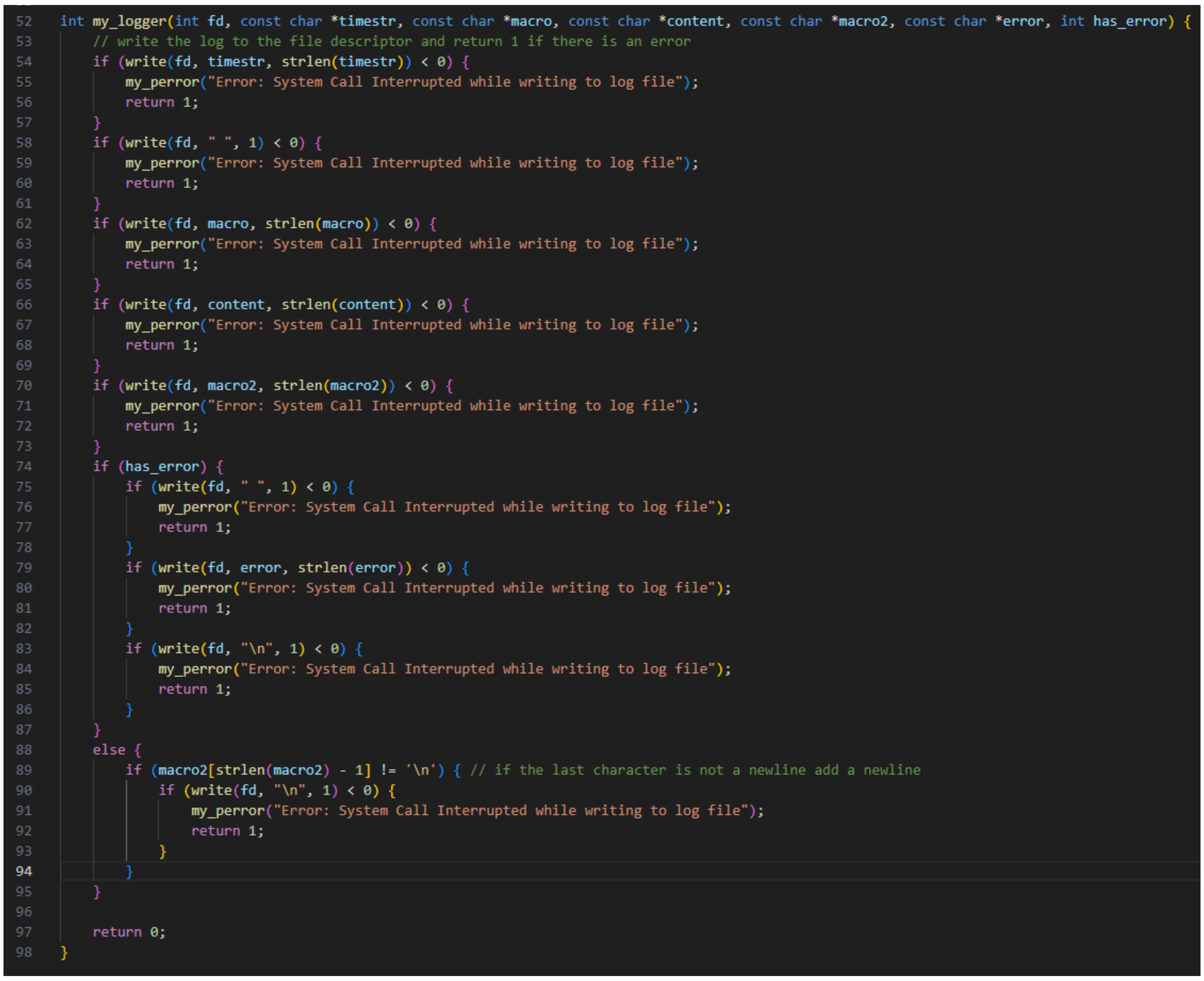


```c
int add_log(const char *macro, const char *content, const char *macro2, const char *error_str, int has_error) {
    const char *file_name = "log.txt";

    if (macro == NULL) {
        // No need to log this because it is control for the functions general flow
        write(STDERR_FILENO, "Log Macro Cannot be NULL!", strlen("Log Macro Cannot be NULL!"));
        return 1;
    }

    if (content == NULL) {
        // No need to log this because it is control for the functions general flow
        write(STDERR_FILENO, "Log Content Cannot be NULL!", strlen("Log Content Cannot be NULL!"));
        return 1;
    }

    if (macro2 == NULL) {
        // No need to log this because it is control for the functions general flow
        write(STDERR_FILENO, "Log Macro Cannot be NULL!", strlen("Log Macro Cannot be NULL!"));
        return 1;
    }
    if (has_error && error_str == NULL) {
        // No need to log this because it is control for the functions general flow
        write(STDERR_FILENO, "Log Error Cannot be NULL!", strlen("Log Error Cannot be NULL!"));
        return 1;
    }

    // Open the log file, if it does not exist, create it if it exists, append to it
    int fd = open(file_name, O_CREAT | O_WRONLY | O_APPEND, 0644);

    if (fd == -1) { // open error
        write(STDERR_FILENO, "Error: Cannot open log file!", strlen("Error: Cannot open log file!"));
        return 1;
    }

    time_t now;
    struct tm *time_info;
    char time_str[22]; // fixed size for time string "[YYYY-MM-DD HH:MM:SS]"

    if (time(&now) < 0){ // time() returns -1 on error
        my_perror("Error Cannot get time while logging: ");
        if (close(fd) == -1) {
            my_perror("Error Cannot close log file: ");
        }
        return 1;
    }

```



```c
    time_t now;
    struct tm *time_info;
    char time_str[22]; // fixed size for time string "[YYYY-MM-DD HH:MM:SS]"

    if (time(&now) < 0){ // time() returns -1 on error
        my_perror("Error Cannot get time while logging: ");
        if (close(fd) == -1) {
            my_perror("Error Cannot close log file: ");
        }
        return 1;
    }

    time_info = localtime(&now);

    // strftime() returns 0 on error
    if (strftime(time_str, sizeof(time_str), "[%Y-%m-%d %H:%M:%S]", time_info) == 0) {
        // strftime is not a system call so it has no errno
        write(STDERR_FILENO, "Error: Cannot get time while logging!", strlen("Error: Cannot get time while logging!"));
        if (close(fd) == -1) {
            my_perror("Error Cannot close log file: ");
        }
        return 1;
    }

    if (my_logger(fd, time_str, macro, content, macro2, error_str, has_error) == 1) {
        if (close(fd) == -1) {
            my_perror("Error Cannot close log file: ");
        }
        return 1;
    }
    if (close(fd) == -1) {
        my_perror("Error Cannot close log file: ");
        return 1;
    }

    return 0;
}
```

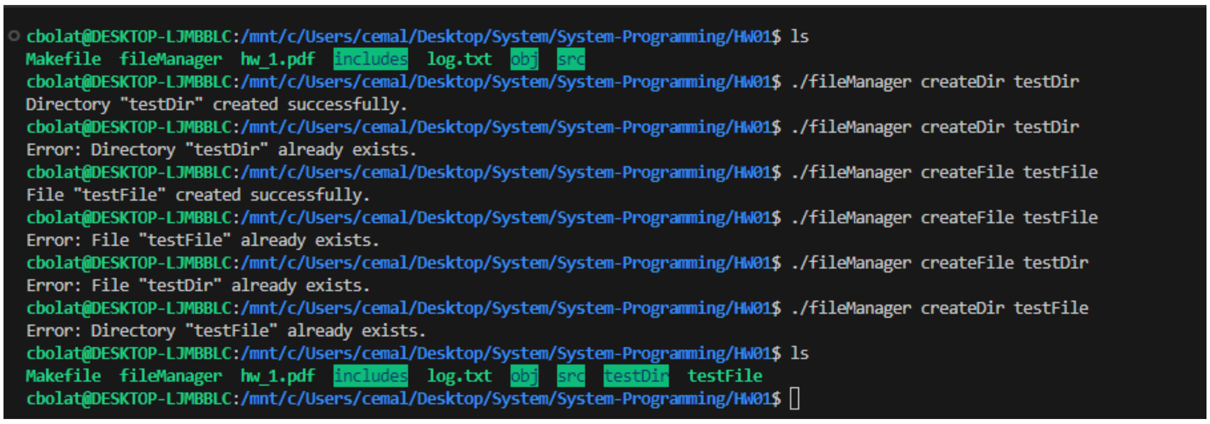
my_logger function is responsible for writing the log entry to the log file. It takes the file descriptor (fd), the formatted timestamp (timestr), the log message components (macro, content, macro2), and optionally the error message (error_str) if an error occurred. It writes each part of the log to the file, ensuring the message is properly formatted. If has_error is true, it includes the error message in the log and appends a newline at the end. If no error occurs, it ensures that the log entry ends with a newline. The function returns 0 if all write operations are successful or 1 if any write operation fails.



```c
int my_logger(int fd, const char *timestr, const char *macro, const char *content, const char *macro2, const char *error, int has_error) {
    // write the log to the file descriptor and return 1 if there is an error
    if (write(fd, timestr, strlen(timestr)) < 0) {
        my_perror("Error: System Call Interrupted while writing to log file");
        return 1;
    }
    if (write(fd, " ", 1) < 0) {
        my_perror("Error: System Call Interrupted while writing to log file");
        return 1;
    }
    if (write(fd, macro, strlen(macro)) < 0) {
        my_perror("Error: System Call Interrupted while writing to log file");
        return 1;
    }
    if (write(fd, content, strlen(content)) < 0) {
        my_perror("Error: System Call Interrupted while writing to log file");
        return 1;
    }
    if (write(fd, macro2, strlen(macro2)) < 0) {
        my_perror("Error: System Call Interrupted while writing to log file");
        return 1;
    }
    if (has_error) {
        if (write(fd, " ", 1) < 0) {
            my_perror("Error: System Call Interrupted while writing to log file");
            return 1;
        }
        if (write(fd, error, strlen(error)) < 0) {
            my_perror("Error: System Call Interrupted while writing to log file");
            return 1;
        }
        if (write(fd, "\n", 1) < 0) {
            my_perror("Error: System Call Interrupted while writing to log file");
            return 1;
        }
    }
    else {
        if (macro2[strlen(macro2) - 1] != '\n') { // if the last character is not a newline add a newline
            if (write(fd, "\n", 1) < 0) {
                my_perror("Error: System Call Interrupted while writing to log file");
                return 1;
            }
        }
    }

    return 0;
}
```

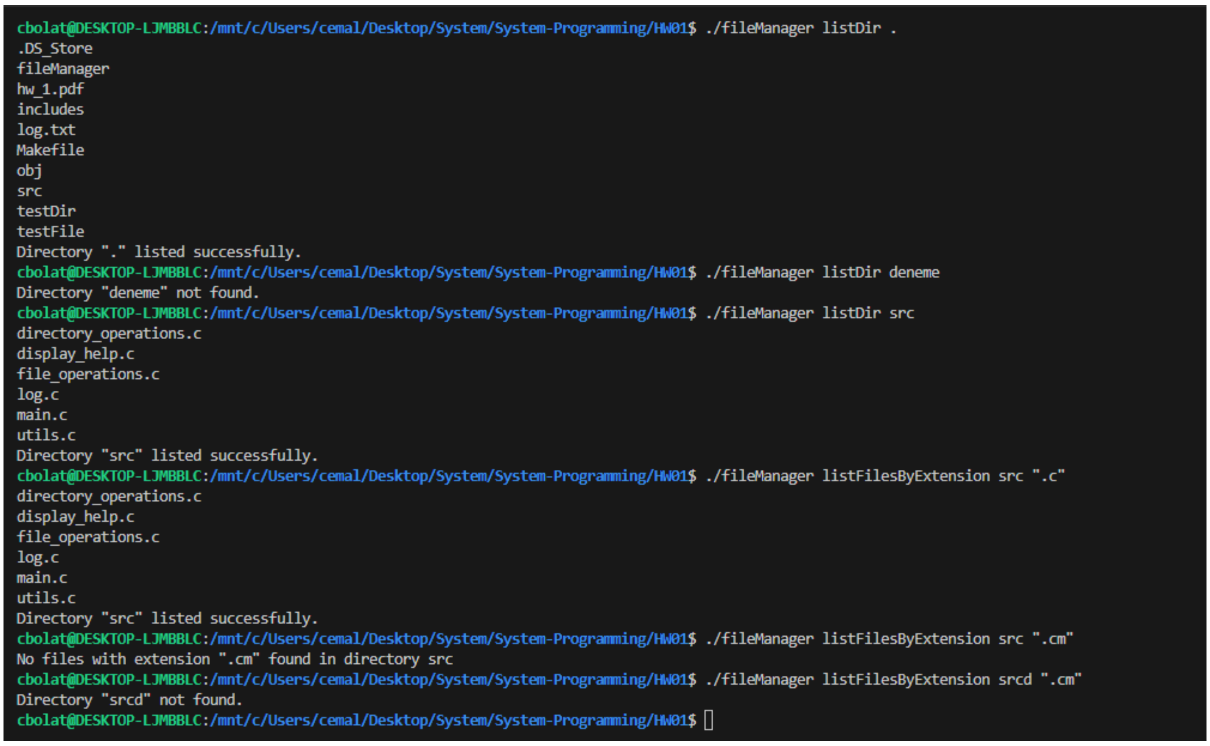
SCREENSHOTS:

create_file() & create_dir()



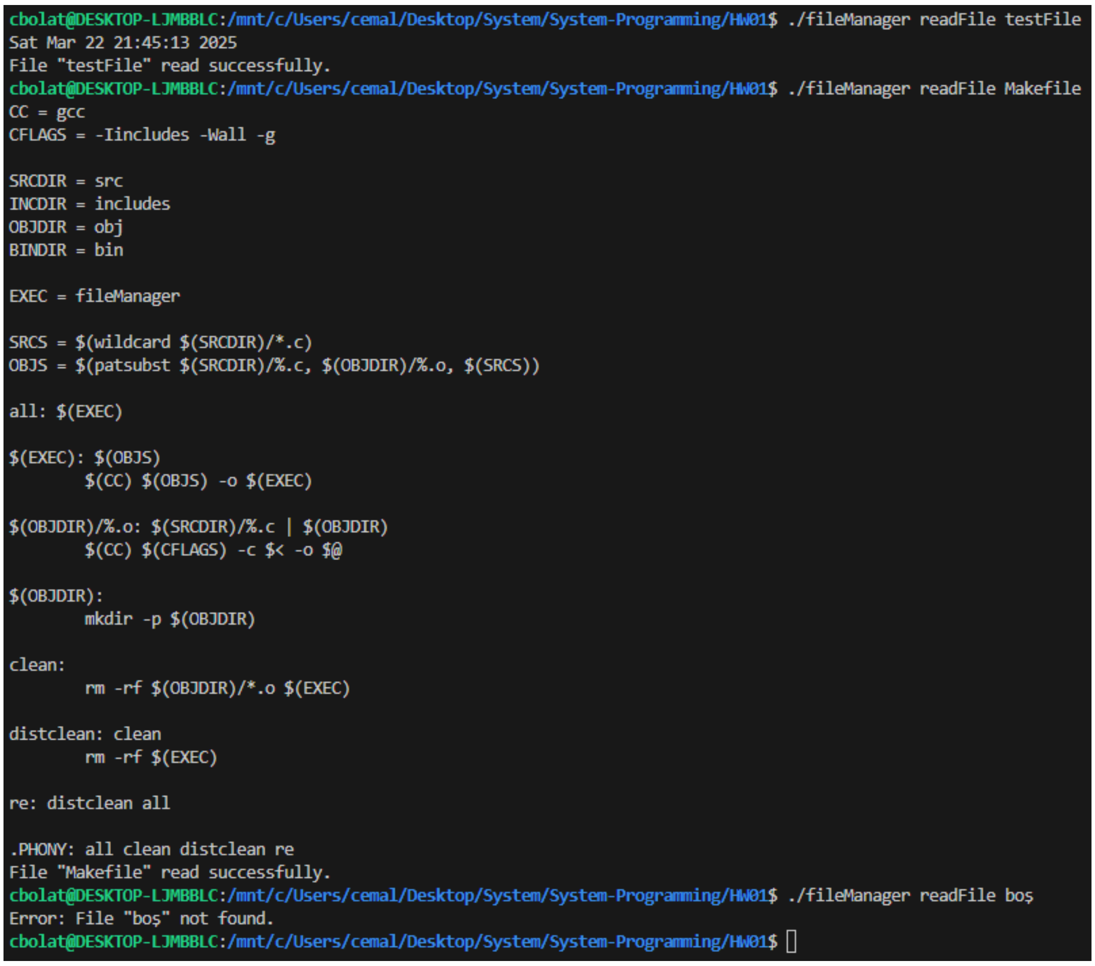
```
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ls
Makefile  fileManager  hw_1.pdf  includes  log.txt  obj  src
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager createDir testDir
Directory "testDir" created successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager createDir testDir
Error: Directory "testDir" already exists.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager createFile testFile
File "testFile" created successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager createFile testFile
Error: File "testFile" already exists.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager createFile testDir
Error: File "testDir" already exists.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager createDir testFile
Error: Directory "testFile" already exists.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ls
Makefile  fileManager  hw_1.pdf  includes  log.txt  obj  src  testDir  testFile
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ 
```

list_dir() & listFilesByExtension()



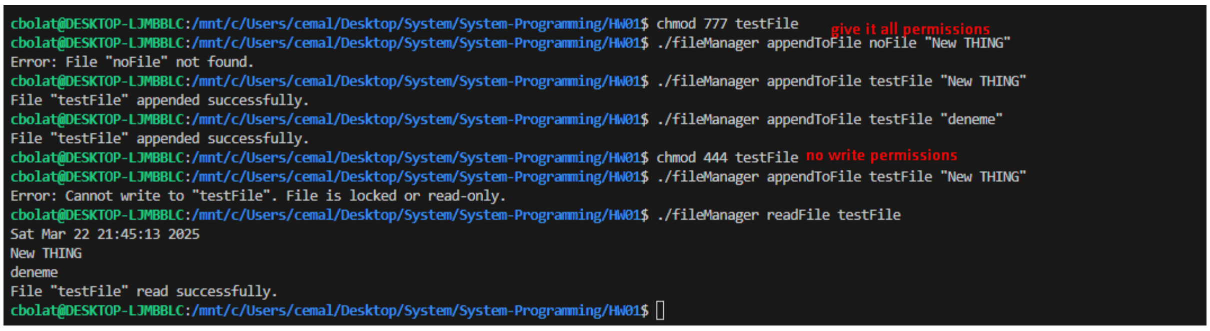
```
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager listDir .
.DS_Store
fileManager
hw_1.pdf
includes
log.txt
Makefile
obj
src
testDir
testFile
Directory "." listed successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager listDir deneme
Directory "deneme" not found.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager listDir src
directory_operations.c
display_help.c
file_operations.c
log.c
main.c
utils.c
Directory "src" listed successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager listFilesByExtension src ".c"
directory_operations.c
display_help.c
file_operations.c
log.c
main.c
utils.c
Directory "src" listed successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager listFilesByExtension src ".cm"
No files with extension ".cm" found in directory src
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager listFilesByExtension srcd ".cm"
Directory "srcd" not found.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ 
```

read_file()



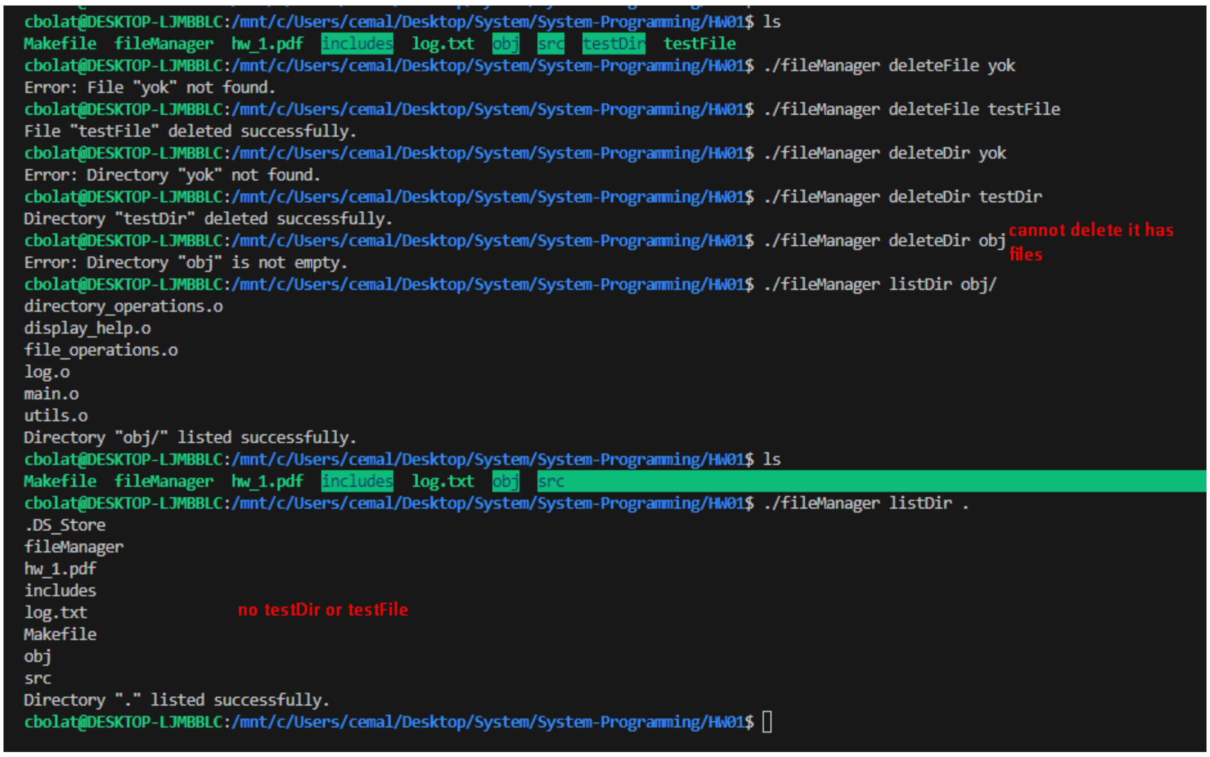
```
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager readFile testFile
Sat Mar 22 21:45:13 2025
File "testFile" read successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager readFile Makefile
CC = gcc
CFLAGS = -Iincludes -Wall -g

SRCDIR = src
INCDIR = includes
OBJDIR = obj
BINDIR = bin

EXEC = fileManager

SRCS = $(wildcard $(SRCDIR)/*.c)
OBJS = $(patsubst $(SRCDIR)/%.c, $(OBJDIR)/%.o, $(SRCS))

all: $(EXEC)

$(EXEC): $(OBJS)
        $(CC) $(OBJS) -o $(EXEC)

$(OBJDIR)/%.o: $(SRCDIR)/%.c | $(OBJDIR)
        $(CC) $(CFLAGS) -c $< -o $@

$(OBJDIR):
        mkdir -p $(OBJDIR)

clean:
        rm -rf $(OBJDIR)/*.o $(EXEC)

distclean: clean
        rm -rf $(EXEC)

re: distclean all

.PHONY: all clean distclean re
File "Makefile" read successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager readFile boş
Error: File "boş" not found.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ 
```

append_to_file():



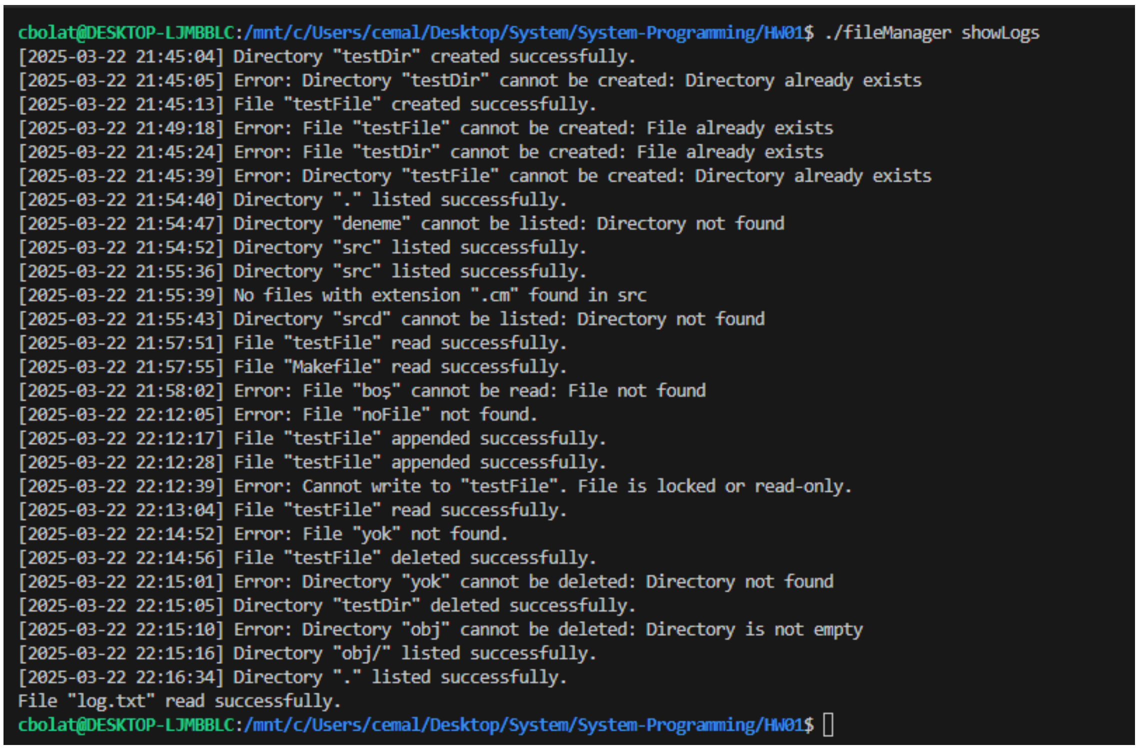
```
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ chmod 777 testFile    give it all permissions
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager appendToFile noFile "New THING"
Error: File "noFile" not found.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager appendToFile testFile "New THING"
File "testFile" appended successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager appendToFile testFile "deneme"
File "testFile" appended successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ chmod 444 testFile no write permissions
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager appendToFile testFile "New THING"
Error: Cannot write to "testFile". File is locked or read-only.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager readFile testFile
Sat Mar 22 21:45:13 2025
New THING
deneme
File "testFile" read successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ 
```

deleteFile() & deleteDir()



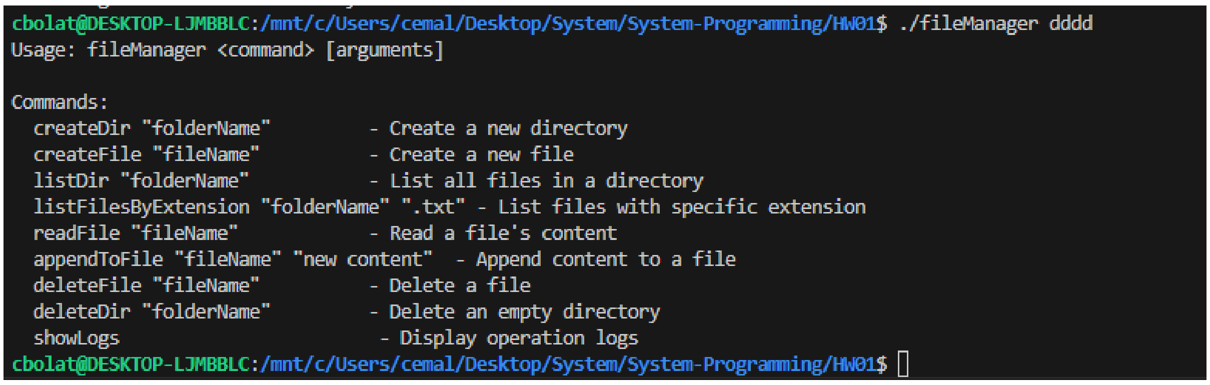
```
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ls
Makefile  fileManager  hw_1.pdf  includes  log.txt  obj  src  testDir  testFile
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager deleteFile yok
Error: File "yok" not found.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager deleteFile testFile
File "testFile" deleted successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager deleteDir yok
Error: Directory "yok" not found.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager deleteDir testDir
Directory "testDir" deleted successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager deleteDir obj
Error: Directory "obj" is not empty.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager listDir obj/
directory_operations.o
display_help.o
file_operations.o
log.o
main.o
utils.o
Directory "obj/" listed successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ls
Makefile  fileManager  hw_1.pdf  includes  log.txt  obj  src
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager listDir .
.DS_Store
fileManager
hw_1.pdf
includes
log.txt
Makefile
obj
src
Directory "." listed successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$
```

ShowLogs



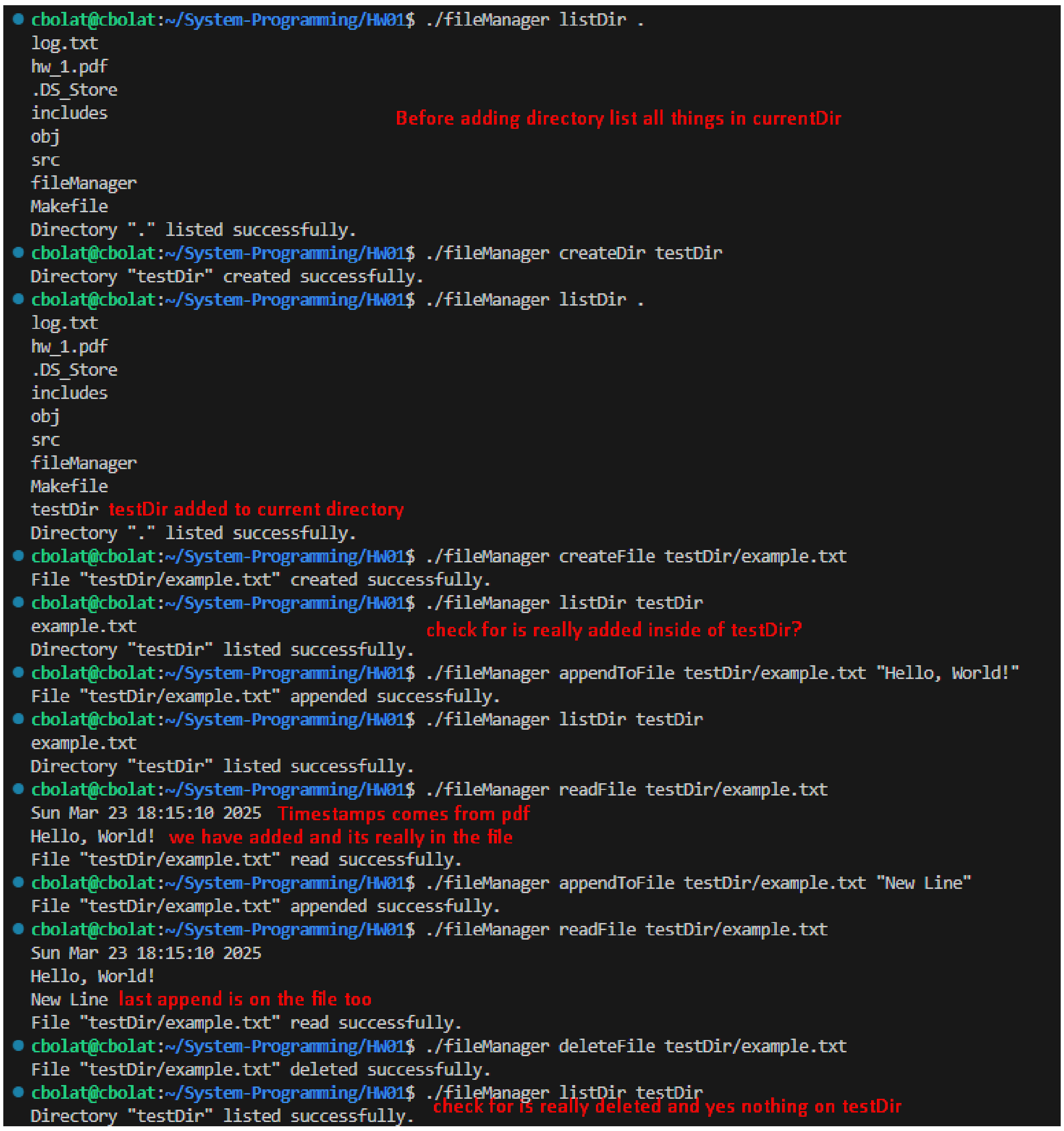
```
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager showLogs
[2025-03-22 21:45:04] Directory "testDir" created successfully.
[2025-03-22 21:45:05] Error: Directory "testDir" cannot be created: Directory already exists
[2025-03-22 21:45:13] File "testFile" created successfully.
[2025-03-22 21:49:18] Error: File "testFile" cannot be created: File already exists
[2025-03-22 21:45:24] Error: File "testDir" cannot be created: File already exists
[2025-03-22 21:45:39] Error: Directory "testFile" cannot be created: Directory already exists
[2025-03-22 21:54:40] Directory "." listed successfully.
[2025-03-22 21:54:47] Directory "deneme" cannot be listed: Directory not found
[2025-03-22 21:54:52] Directory "src" listed successfully.
[2025-03-22 21:55:36] Directory "src" listed successfully.
[2025-03-22 21:55:39] No files with extension ".cm" found in src
[2025-03-22 21:55:43] Directory "srcd" cannot be listed: Directory not found
[2025-03-22 21:57:51] File "testFile" read successfully.
[2025-03-22 21:57:55] File "Makefile" read successfully.
[2025-03-22 21:58:02] Error: File "boş" cannot be read: File not found
[2025-03-22 22:12:05] Error: File "noFile" not found.
[2025-03-22 22:12:17] File "testFile" appended successfully.
[2025-03-22 22:12:28] File "testFile" appended successfully.
[2025-03-22 22:12:39] Error: Cannot write to "testFile". File is locked or read-only.
[2025-03-22 22:13:04] File "testFile" read successfully.
[2025-03-22 22:14:52] Error: File "yok" not found.
[2025-03-22 22:14:56] File "testFile" deleted successfully.
[2025-03-22 22:15:01] Error: Directory "yok" cannot be deleted: Directory not found
[2025-03-22 22:15:05] Directory "testDir" deleted successfully.
[2025-03-22 22:15:10] Error: Directory "obj" cannot be deleted: Directory is not empty
[2025-03-22 22:15:16] Directory "obj/" listed successfully.
[2025-03-22 22:16:34] Directory "." listed successfully.
File "log.txt" read successfully.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$
```

Helper



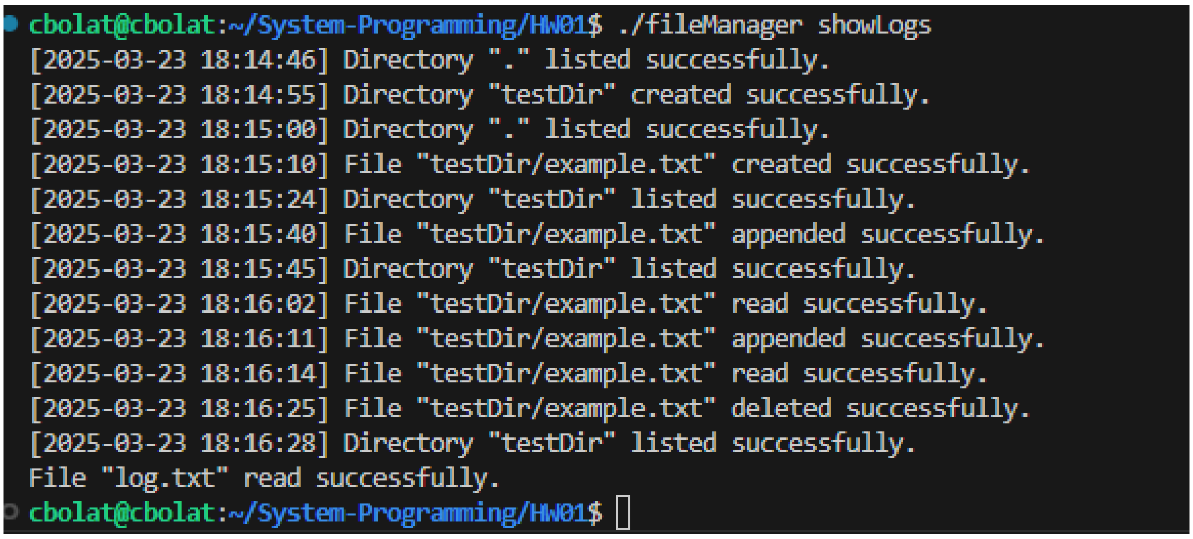
```
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$ ./fileManager dddd
Usage: fileManager <command> [arguments]

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
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System/System-Programming/HW01$
```

TESTING SCENERIO FROM PDF



```
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager listDir .
  log.txt
  hw_1.pdf
  .DS_Store
  includes                                Before adding directory list all things in currentDir
  obj
  src
  fileManager
  Makefile
  Directory "." listed successfully.
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager createDir testDir
  Directory "testDir" created successfully.
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager listDir .
  log.txt
  hw_1.pdf
  .DS_Store
  includes
  obj
  src
  fileManager
  Makefile
  testDir testDir added to current directory
  Directory "." listed successfully.
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager createFile testDir/example.txt
  File "testDir/example.txt" created successfully.
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager listDir testDir
  example.txt                              check for is really added inside of testDir?
  Directory "testDir" listed successfully.
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager appendToFile testDir/example.txt "Hello, World!"
  File "testDir/example.txt" appended successfully.
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager listDir testDir
  example.txt
  Directory "testDir" listed successfully.
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager readFile testDir/example.txt
  Sun Mar 23 18:15:10 2025  Timestamps comes from pdf
  Hello, World!  we have added and its really in the file
  File "testDir/example.txt" read successfully.
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager appendToFile testDir/example.txt "New Line"
  File "testDir/example.txt" appended successfully.
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager readFile testDir/example.txt
  Sun Mar 23 18:15:10 2025
  Hello, World!
  New Line last append is on the file too
  File "testDir/example.txt" read successfully.
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager deleteFile testDir/example.txt
  File "testDir/example.txt" deleted successfully.
● cbolat@cbolat:~/System-Programming/HW01$ ./fileManager listDir testDir
  Directory "testDir" listed successfully. check for is really deleted and yes nothing on testDir
```

Logs on next page



```
cbolat@cbolat:~/System-Programming/HW01$ ./fileManager showLogs
[2025-03-23 18:14:46] Directory "." listed successfully.
[2025-03-23 18:14:55] Directory "testDir" created successfully.
[2025-03-23 18:15:00] Directory "." listed successfully.
[2025-03-23 18:15:10] File "testDir/example.txt" created successfully.
[2025-03-23 18:15:24] Directory "testDir" listed successfully.
[2025-03-23 18:15:40] File "testDir/example.txt" appended successfully.
[2025-03-23 18:15:45] Directory "testDir" listed successfully.
[2025-03-23 18:16:02] File "testDir/example.txt" read successfully.
[2025-03-23 18:16:11] File "testDir/example.txt" appended successfully.
[2025-03-23 18:16:14] File "testDir/example.txt" read successfully.
[2025-03-23 18:16:25] File "testDir/example.txt" deleted successfully.
[2025-03-23 18:16:28] Directory "testDir" listed successfully.
File "log.txt" read successfully.
cbolat@cbolat:~/System-Programming/HW01$ 
```

Conclusion:

Throughout this implementation, one of the major challenges I faced was handling everything with system calls, such as write(), open(), and close(), which require working at a lower level than typical high-level functions. This approach demands a greater attention to error handling, as each system call can fail in multiple ways, and the code needs to account for these failures to ensure robust behavior. Managing file operations manually—like opening log files, writing log entries, and properly closing them—required a deeper understanding of system behavior, which also made debugging more challenging.

To address these difficulties, I developed several helper functions, such as my_write, my_perror, and my_logger, to encapsulate repetitive logic and improve code readability. These functions allowed me to handle errors centrally and make the overall implementation more manageable, ensuring that the same pattern was followed consistently. This modular approach helped streamline the process and reduce the complexity of error-prone tasks, such as writing logs or managing system errors.

Through research and practice, I became more comfortable with manually using system calls. This experience not only enhanced my understanding of low-level operations but also improved my skills in managing more complex manual processes. It also pushed me to think critically about how best to structure code for reliability, especially in error-prone environments like file I/O. In the end, this project provided valuable hands-on experience with system-level programming and helped solidify my ability to handle more manual, low-level tasks effectively.