Secure File and Directory Management System

CSE 344 - System Programming

Homework #1

Name: Ziya Kadir TOKLUOĞLU

Student ID: 210104004228

Date: March 23, 2025

Instructor: Z. Bilici

Submission Date: 23 March 23:59

Contents

1 Introduction

This project implements a Secure File and Directory Management System in C, utilizing Linux system calls. The system supports various file and directory operations, such as creation, deletion, and listing.

The creation of processes using `fork()` is a key component of this implementation. Child processes are responsible for handling file and directory operations, while the parent process manages logging activities. Child processes communicate the success or failure of operations to the parent, who then records the results in `log.txt`. This ensures that operations are properly documented without interfering with the execution flow.

Another crucial aspect of the system is the exclusive use of system calls for all operations, avoiding high-level abstractions. This approach provides a deeper understanding of system programming methodologies by directly interacting with the Linux kernel.

Furthermore, a Bash-written test script automates the execution of predefined test scenarios, ensuring correctness and consistency of the implemented functionality. This script verifies that all commands function as expected and logs the outcomes systematically.

2 Implementation Details

2.1 Listing Files by Extension

```c
#include "../../directory_management.h"


int list_files_by_extension(const char *folder_name, const char *extension) {
    pid_t pid = fork();

    if (pid < 0) {
        write(STDERR_FILENO, "Fork␣failed\n", 12);
        log_output("Error:␣fork␣failed␣when␣listing␣files␣by␣extension␣in␣\"%s\".", folder_name);
        return -1;
    }

    if (pid == 0) {  // Child process: Handles file listing
        DIR *dir = opendir(folder_name);
        if (!dir) {
            write(STDOUT_FILENO, "Error:␣Directory␣not␣found.\n", 28);
            exit(1);  // Exit code 1: Directory not found
        }

        struct dirent *entry;
        int found = 0;
        struct stat st;
        char filepath[512];

        while ((entry = readdir(dir)) != NULL) {
            strcpy(filepath, folder_name);
            strcat(filepath, "/");
            strcat(filepath, entry->d_name);

            if (stat(filepath, &st) == 0 && S_ISREG(st.st_mode)) {
                const char *ext = strrchr(entry->d_name, '.');
                if (ext && strcmp(ext, extension) == 0) {
                    write(STDOUT_FILENO, entry->d_name, strlen(entry->d_name));
                    write(STDOUT_FILENO, "\n", 1);
                    found = 1;
                }
            }
        }

        closedir(dir);

        if (found)
            exit(0);
        else
            exit(2);
    } else {
        int status;
        if (waitpid(pid, &status, 0) == -1) {
            write(STDERR_FILENO, "waitpid␣failed\n", 15);
            log_output("Error:␣waitpid␣failed␣when␣listing␣
               files␣in␣\"%s\".", folder_name);
            return -1;
        }

        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);

            if (exit_status == 0) {
                log_output("Files␣with␣extension␣\"%s\"␣
                   listed␣successfully␣in␣\"%s\".", extension
                   , folder_name);
                return 0;
            } else if (exit_status == 2) {
                log_output("No␣files␣with␣extension␣\"%s\"␣
                   found␣in␣\"%s\".", extension, folder_name)
                   ;
                write(STDOUT_FILENO, "No␣files␣with␣extension
                   ␣\"", 25);
                write(STDOUT_FILENO, extension, strlen(
                   extension));
                write(STDOUT_FILENO, "\"␣found␣in␣\"", 11);
                write(STDOUT_FILENO, folder_name, strlen(
                   folder_name));
                write(STDOUT_FILENO, "\".\n", 3);
                return -1;
            } else {
                log_output("Error:␣Unable␣to␣access␣directory
                   ␣\"%s\".", folder_name);
                write(STDOUT_FILENO, "Error:␣Unable␣to␣access
                   ␣directory␣\"", 35);
                write(STDOUT_FILENO, folder_name, strlen(
                   folder_name));
                write(STDOUT_FILENO, "\".\n", 3);
                return -1;
            }
        } else {
            log_output("Error:␣Child␣terminated␣abnormally␣
               "while␣listing␣files␣in␣\"%s\".", folder_name);
            return -1;
        }
    }
}
```

The function `list_files_by_extension()` is responsible for listing all files with a specific extension in a given directory. It utilizes the following key system calls:

- `fork()` - This system call is used to create a child process that will handle the directory listing separately. The parent process waits for the child's completion.
- `opendir()` - Opens the specified directory and returns a pointer to a `DIR` structure, which is used for iterating through the directory's contents.
- `readdir()` - Reads directory entries one by one.
- `stat()` - Retrieves information about the file, allowing us to determine if it is a regular file.
- `write()` - Outputs filenames directly to `STDOUT_FILENO` (standard output).
- `waitpid()` - Ensures that the parent process waits for the child process to complete before proceeding.

The function iterates over all entries in the directory and checks if each entry is a regular file (`S_ISREG(st.st_mode)`). If the file matches the given extension, it is printed. The exit status of the child process is used to handle different cases, such as directory not found, no matching files, or successful execution.

2.2 Deleting a Directory

```
#include "../../directory_management.h"


int delete_directory(const char *folder_name) {
    pid_t pid = fork();

    if (pid < 0) {
        write(STDERR_FILENO, "Fork␣failed:␣", 13);
```

```c
        write(STDERR_FILENO, strerror(errno), strlen(strerror(errno)));
        write(STDERR_FILENO, "\n", 1);
        log_output("Error: fork failed when deleting directory \"%s\".", folder_name);
        return -1;
    }

    if (pid == 0) {
        if (rmdir(folder_name) == -1) {
            if (errno == ENOTEMPTY) {
                exit(2);
            } else if (errno == ENOENT) {
                exit(3);
            } else {
                exit(1);
            }
        }
        exit(0);
    } else {
        int status;
        if (waitpid(pid, &status, 0) == -1) {
            write(STDERR_FILENO, "waitpid failed: ", 16);
            write(STDERR_FILENO, strerror(errno), strlen(strerror(errno)));
            write(STDERR_FILENO, "\n", 1);
            log_output("Error: waitpid failed when deleting directory \"%s\".", folder_name);
            return -1;
        }

        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);

            if (exit_status == 0) {
                log_output("Directory \"%s\" deleted successfully.", folder_name);
                return 0;
            } else if (exit_status == 2) {
                log_output("Error: Directory \"%s\" is not empty.", folder_name);
                write(STDOUT_FILENO, "Error: Directory \"", 18);
                write(STDOUT_FILENO, folder_name, strlen(folder_name));
                write(STDOUT_FILENO, "\" is not empty.\n", 16);
                return -1;
            } else if (exit_status == 3) {
                log_output("Error:␣Directory␣\"%s\"␣not␣found.", folder_name);
                write(STDOUT_FILENO, "Error:␣Directory␣\"", 18);
                write(STDOUT_FILENO, folder_name, strlen(folder_name));
                write(STDOUT_FILENO, "\"␣not␣found.\n", 13);
                return -1;
            } else {
                log_output("Error:␣Unable␣to␣delete␣directory␣\"%s\".", folder_name);
                write(STDOUT_FILENO, "Error:␣Unable␣to␣delete␣directory␣\"", 35);
                write(STDOUT_FILENO, folder_name, strlen(folder_name));
                write(STDOUT_FILENO, "\".\n", 3);
                return -1;
            }
        } else {
            log_output("Error:␣Child␣terminated␣abnormally␣while␣deleting␣directory␣\"%s\".", folder_name);
            return -1;
        }
    }
}
```

The `delete_directory()` function deletes a specified directory using the following system calls:

- `fork()` - Creates a child process to handle the deletion.
- `rmdir()` - Removes an empty directory.
- `strerror()` - Retrieves error messages in case the removal fails.
- `waitpid()` - The parent process waits for the child process to complete.
- `write()` - Prints error messages or success confirmation.

If the directory is not empty, the function returns an error code. The `waitpid()` ensures proper synchronization between parent and child processes.

2.3 Creating a Directory

```
#include "../../directory_management.h"

```

```
int create_directory(const char *folder_name) {
    write(STDOUT_FILENO, "Creating␣directory␣\"", 21);
    write(STDOUT_FILENO, folder_name, strlen(folder_name));
    write(STDOUT_FILENO, "\"...\n", 5);

    pid_t pid = fork();

    if (pid < 0) {
        write(STDERR_FILENO, "Fork␣failed:␣", 13);
        write(STDERR_FILENO, strerror(errno), strlen(strerror(errno)));
        write(STDERR_FILENO, "\n", 1);
        log_output("Error:␣fork␣failed␣when␣creating␣directory␣\"%s\".", folder_name);
        return -1;
    }

    if (pid == 0) {
        if (mkdir(folder_name, 0755) == -1) {
            if (errno == EEXIST)
                exit(2);
            else
                exit(1);
        }
        exit(0);
    } else {
        int status;
        if (waitpid(pid, &status, 0) == -1) {
            write(STDERR_FILENO, "waitpid␣failed:␣", 16);
            write(STDERR_FILENO, strerror(errno), strlen(strerror(errno)));
            write(STDERR_FILENO, "\n", 1);
            log_output("Error:␣waitpid␣failed␣when␣creating␣directory␣\"%s\".", folder_name);
            return -1;
        }

        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);

            if (exit_status == 0) {
                log_output("Directory␣\"%s\"␣created␣successfully.", folder_name);
                return 0;
            } else if (exit_status == 2) {
                log_output("Error:␣Directory␣\"%s\"␣already␣exists.", folder_name);
                write(STDOUT_FILENO, "Error:␣Directory␣\"", 18);
                write(STDOUT_FILENO, folder_name, strlen(folder_name));
                write(STDOUT_FILENO, "\"␣already␣exists.\n", 18);
                return -1;
            } else {
                log_output("Error:␣Unable␣to␣create␣directory␣\"%s\".", folder_name);
                write(STDOUT_FILENO, "Error:␣Unable␣to␣create␣directory␣\"", 35);
                write(STDOUT_FILENO, folder_name, strlen(folder_name));
                write(STDOUT_FILENO, "\".\n", 3);
                return -1;
            }
        } else {
            log_output("Error:␣Child␣terminated␣abnormally␣while␣creating␣\"%s\".", folder_name);
            return -1;
        }
    }
}
```

The create_directory() function is used to create a new directory and is implemented using:

- fork() - Creates a child process to perform the operation.
- mkdir() - Creates a directory with 0755 permissions.
- errno - Checks if the directory already exists.
- waitpid() - Ensures that the parent waits for the child process.

If the directory already exists, an error is returned. Otherwise, the directory is successfully created.

2.4 Listing Directory Contents

```
#include "../../directory_management.h"

int list_directory(const char *folder_name) {
    pid_t pid = fork();

    if (pid < 0) {
        write(STDERR_FILENO, "Fork␣failed\n", 12);
        log_output("Error:␣fork␣failed␣when␣listing␣directory␣\"%s\".", folder_name);
```

```c
        return -1;
    }

    if (pid == 0) {
        DIR *dir = opendir(folder_name);
        if (!dir) {
            write(STDOUT_FILENO, "Error:␣Directory␣not␣found.\n", 28);
            exit(1);
        }

        struct dirent *entry;
        while ((entry = readdir(dir)) != NULL) {
            write(STDOUT_FILENO, entry->d_name, strlen(entry->d_name));
            write(STDOUT_FILENO, "\n", 1);
        }

        closedir(dir);
        exit(0);
    } else {
        int status;
        if (waitpid(pid, &status, 0) == -1) {
            write(STDERR_FILENO, "waitpid␣failed\n", 15);
            log_output("Error:␣waitpid␣failed␣when␣listing␣directory␣\"%s\".", folder_name);
            return -1;
        }

        if (WIFEXITED(status) && WEXITSTATUS(status) == 0) {
            log_output("Directory␣\"%s\"␣listed␣successfully.", folder_name);
            return 0;
        } else {
            log_output("Error:␣Directory␣\"%s\"␣not␣found.", folder_name);
            return -1;
        }
    }
}
```

The `list_directory()` function lists the contents of a given directory. It relies on the following system calls:

- `fork()` - Creates a child process for listing.
- `opendir()` - Opens the directory.
- `readdir()` - Iterates over the directory entries.

- `write()` - Prints the directory entries to standard output.
- `waitpid()` - Waits for the child process to finish execution.

If the directory does not exist, an error message is printed. Otherwise, all file and directory names are displayed.

2.5 Appending Content to a File

```c
#include "../../directory_management.h"


int append_to_file(const char *file_name, const char *content) {
    pid_t pid = fork();

    if (pid < 0) {
        write(STDERR_FILENO, "Fork␣failed\n", 12);
        log_output("Error:␣fork␣failed␣when␣appending␣to␣file␣\"%s\".", file_name);
        return -1;
    }

    if (pid == 0) { // Child process performs file appending
        int fd = open(file_name, O_WRONLY | O_APPEND);
        if (fd == -1) {
            write(STDERR_FILENO, "Error␣opening␣file\n", 19);
            exit(1);
        }

        struct flock lock;
        lock.l_type = F_WRLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start = 0;
        lock.l_len = 0;


        if (fcntl(fd, F_SETLK, &lock) == -1) {
            write(STDERR_FILENO, "File␣is␣locked\n", 15);
            close(fd);
            exit(2);
        }

        if (write(fd, content, strlen(content)) < 0) {
            write(STDERR_FILENO, "Error␣writing␣to␣file\n", 22);
            lock.l_type = F_UNLCK;
            fcntl(fd, F_SETLK, &lock);
            close(fd);
            exit(1);
        }


        lock.l_type = F_UNLCK;
        fcntl(fd, F_SETLK, &lock);
        close(fd);
        exit(0);
    } else {
        int status;
        waitpid(pid, &status, 0);

        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);

            if (exit_status == 0) {
                log_output("Content␣appended␣successfully␣to␣
                   \"%s\".", file_name);
                return 0;
            } else if (exit_status == 2) {
                log_output("Error:␣Cannot␣write␣to␣\"%s\".␣
                   File␣is␣locked.", file_name);
                return -1;
            } else {
                log_output("Error:␣Cannot␣write␣to␣\"%s\".␣
                   File␣is␣locked␣or␣read-only.", file_name);
                return -1;
            }
        } else {
            log_output("Error:␣Child␣terminated␣abnormally␣
               while␣appending␣to␣\"%s\".", file_name);
            return -1;
        }
    }
}
```

The `append_to_file()` function appends data to an existing file. It uses:

- `fork()` - Creates a separate process for appending.
- `open()` - Opens the file in append mode (`O_WRONLY | O_APPEND`).
- `fcntl()` - Implements file locking to prevent concurrent writes.
- `write()` - Appends the new content.
- `close()` - Closes the file after writing.

If the file is locked by another process, an error message is returned.

2.6 Creating a File

```c
#include "../../directory_management.h"



int create_file(const char *file_name) {
    pid_t pid = fork();

    if (pid < 0) {
        write(STDERR_FILENO, "Fork␣failed\n", 12);
        log_output("Error:␣fork␣failed␣when␣creating␣file␣\"%s\".", file_name);
        return -1;
    }

    if (pid == 0) {
        int fd = open(file_name, O_WRONLY | O_CREAT | O_EXCL, 0644);
        if (fd == -1) {
            if (errno == EEXIST) {
                exit(2);
            } else {
                exit(1);
            }
        }

        char *timestamp = get_timestamp();
        if (!timestamp) {
            close(fd);
            exit(1);
        }

        if (write(fd, timestamp, strlen(timestamp)) < 0) {
            write(STDERR_FILENO, "Error␣writing␣timestamp␣to␣file\n", 32);
            close(fd);
            free(timestamp);
            exit(1);
        }

        free(timestamp);
        close(fd);
        exit(0);
    } else {
        int status;
        if (waitpid(pid, &status, 0) == -1) {
            write(STDERR_FILENO, "waitpid␣failed\n", 15);
            log_output("Error:␣waitpid␣failed␣while␣creating␣
                file␣\"%s\".", file_name);
            return -1;
        }

        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);
            if (exit_status == 0) {
                log_output("File␣\"%s\"␣created␣successfully.", file_name);
                return 0;
            } else if (exit_status == 2) {
                log_output("Error:␣File␣\"%s\"␣already␣exists.", file_name);
                return -1;
            } else {
                log_output("Error:␣Unable␣to␣create␣file␣\"%s\".", file_name);
                return -1;
            }
        } else {
            log_output("Error:␣Child␣terminated␣abnormally␣while␣creating␣file␣\"%s\".", file_name);
            return -1;
        }
    }
}
```

The `create_file()` function creates a new file and writes a timestamp into it. It employs:

- `fork()` - A child process handles file creation.
- `open()` - Opens the file with `O_CREAT | O_EXCL` to ensure it does not already exist.
- `write()` - Writes a timestamp into the file.
- `close()` - Closes the file descriptor.
- `waitpid()` - Ensures proper synchronization.

If the file already exists, an error is returned.

2.7 Reading a File

```
#include "../../directory_management.h"


```

```c
int read_file(const char *file_name) {
    pid_t pid = fork();

    if (pid < 0) {
        write(STDERR_FILENO, "Fork failed\n", 12);
        log_output("Error: fork failed while reading file \"%s\".", file_name);
        return -1;
    }

    if (pid == 0) {
        int fd = open(file_name, O_RDONLY);
        if (fd == -1) {
            write(STDOUT_FILENO, "Error: File \"", 13);
            write(STDOUT_FILENO, file_name, strlen(file_name));
            write(STDOUT_FILENO, "\" not found.\n", 13);
            exit(1);
        }

        char buffer[256];
        ssize_t bytes_read;

        while ((bytes_read = read(fd, buffer, sizeof(buffer))) > 0) {
            write(STDOUT_FILENO, buffer, bytes_read);
        }

        close(fd);
        exit(0);
    } else {
        int status;
        if (waitpid(pid, &status, 0) == -1) {
            write(STDERR_FILENO, "waitpid failed\n", 15);
            log_output("Error: waitpid failed when reading file \"%s\".", file_name);
            return -1;
        }

        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);
            if (exit_status == 0) {
                log_output("File \"%s\" read successfully.", file_name);
                return 0;
            } else {
                log_output("Error: File \"%s\" not found or unable to read.", file_name);
                return -1;
            }
        } else {
            log_output("Error:␣Child␣terminated␣abnormally␣
               while␣reading␣\"%s\".", file_name);
            return -1;
        }
    }
}
```

The read_file() function reads and displays the contents of a file. It makes use of:

- fork() - A child process handles reading.
- open() - Opens the file in read mode (O_RDONLY).
- read() - Reads data into a buffer.
- write() - Outputs the buffer to standard output.
- close() - Closes the file descriptor.

If the file does not exist, an error is displayed.

2.8 Deleting a File

```
#include "../../directory_management.h"


int delete_file(const char *file_name) {
    pid_t pid = fork();

    if (pid < 0) {
        write(STDERR_FILENO, "Fork␣failed\n", 12);
        log_output("Error:␣fork␣failed␣when␣deleting␣file␣\"%
           s\".", file_name);
        return -1;
    }

    if (pid == 0) {
        if (unlink(file_name) == -1) {
            if (errno == ENOENT) {
                exit(2);
            } else {
                exit(1);
            }
        }
        exit(0);
    } else {
        int status;
        if (waitpid(pid, &status, 0) == -1) {
            write(STDERR_FILENO, "waitpid␣failed\n", 15);
            log_output("Error:␣waitpid␣failed␣when␣deleting␣file␣\"%s\".", file_name);
            return -1;
        }

        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);

            if (exit_status == 0) {
                log_output("File␣\"%s\"␣deleted␣successfully.", file_name);
                return 0;
            } else if (exit_status == 2) {
                log_output("Error:␣File␣\"%s\"␣not␣found.", file_name);
                write(STDOUT_FILENO, "Error:␣File␣\"", 13);
                write(STDOUT_FILENO, file_name, strlen(file_name));
                write(STDOUT_FILENO, "\"␣not␣found.\n", 13);
                return -1;
            } else {
                log_output("Error:␣Unable␣to␣delete␣file␣\"%s\".", file_name);
                return -1;
            }
        } else {
            log_output("Error:␣Child␣terminated␣abnormally␣while␣deleting␣\"%s\".", file_name);
            return -1;
        }
    }
}
```

The `delete_file()` function deletes a file using:

- `fork()` - A child process performs deletion.
- `unlink()` - Removes the file.
- `errno` - Handles errors such as ”file not found.”
- `waitpid()` - Ensures that the parent process waits for the child's execution.

If the file does not exist, an error is returned.

2.9 Logging Operations

```c
#include "log_operation.h"

char* get_timestamp() {
    char *timestamp = malloc(TIMESTAMP_SIZE);
    if (!timestamp) {
        write(STDERR_FILENO, "Memory␣allocation␣failed\n",
            25);
        return NULL;
    }

    time_t now = time(NULL);
    struct tm *t = localtime(&now);
    strftime(timestamp, TIMESTAMP_SIZE, "[%Y-%m-%d␣%H:%M:%S]"
        , t);

    return timestamp;
}

void log_operation(const char *message) {
    char *timestamp = get_timestamp();
    if (!timestamp) {
        return;
    }

    int fd = open("logs/log.txt", O_WRONLY | O_CREAT |
        O_APPEND, 0644);
    if (fd < 0) {
        write(STDERR_FILENO, "Error␣opening␣log␣file\n", 23);
        free(timestamp);
        return;
    }


    char buffer[512];
    strcpy(buffer, timestamp);
    strcat(buffer, "␣");
    strcat(buffer, message);
    strcat(buffer, "\n");

    if (write(fd, buffer, strlen(buffer)) < 0) {
        write(STDERR_FILENO, "Error␣writing␣to␣log␣file\n",
            26);
    }

    close(fd);
    free(timestamp);
}
```

```c

void log_output(const char *format, ...) {
    char buffer[512];
    va_list args;
    va_start(args, format);


    char *p = buffer;
    const char *f = format;
    while (*f && (p - buffer) < (long int)sizeof(buffer) - 1) {
        if (*f == '%' && *(f + 1) == 's') {
            f += 2;
            const char *str_arg = va_arg(args, const char *);
            while (*str_arg && (p - buffer) < (long int)sizeof(buffer) - 1) {
                *p++ = *str_arg++;
            }
        } else {
            *p++ = *f++;
        }
    }
    *p = '\0';

    va_end(args);

    log_operation(buffer);
}



int show_logs(void) {
    pid_t pid = fork();

    if (pid < 0) {
        write(STDERR_FILENO, "Fork␣failed\n", 12);
        log_output("Error:␣fork␣failed␣while␣displaying␣logs.");
        return -1;
    }

    if (pid == 0) {
        int fd = open("logs/log.txt", O_RDONLY);
        if (fd < 0) {
            write(STDERR_FILENO, "Error␣opening␣log␣file\n", 23);
            exit(1);
        }

        char buffer[256];
        ssize_t bytes_read;


        while ((bytes_read = read(fd, buffer, sizeof(buffer))) > 0) {
            if (write(STDOUT_FILENO, buffer, bytes_read) < 0) {
                write(STDERR_FILENO, "Error writing logs to stdout\n", 30);
                close(fd);
                exit(1);
            }
        }

        close(fd);
        exit(0);
    } else {
        int status;
        if (waitpid(pid, &status, 0) == -1) {
            write(STDERR_FILENO, "waitpid failed\n", 15);
            log_output("Error: waitpid failed while displaying logs.");
            return -1;
        }

        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);

            if (exit_status == 0) {
                log_output("Logs displayed successfully.");
                return 0;
            } else {
                log_output("Error: Unable to display logs.");
                return -1;
            }
        } else {
            log_output("Error: Child terminated abnormally while displaying logs.");
            return -1;
        }
    }
}
```

The `log_operation.c` file is responsible for handling logging mechanisms in the system. It ensures that every file and directory operation is recorded in a structured log file. This module consists of four primary functions:

- `get_timestamp()` - Generates a formatted timestamp.

- `log_operation()` - Logs messages to a file with timestamps.
- `log_output()` - Supports formatted log messages with variable arguments.
- `show_logs()` - Displays the content of the log file.

Timestamp Generation (`get_timestamp()`) This function generates a timestamp in the format `"[YYYY-MM-DD HH:MM:SS]"` and returns it as a dynamically allocated string. It makes use of:

- `time()` - Retrieves the current system time.
- `localtime()` - Converts raw time into a structured format.
- `strftime()` - Formats the time into a readable string.
- `malloc()` - Allocates memory for the timestamp string.

If memory allocation fails, an error message is written to standard error.

Logging Operations (`log_operation()`) This function appends log messages to `logs/log.txt`, ensuring that all actions performed in the system are recorded. It uses:

- `open()` - Opens the log file with `O_WRONLY | O_CREAT | O_APPEND`, ensuring that logs are appended without overwriting existing data.
- `write()` - Writes the log message with a timestamp.
- `close()` - Closes the file descriptor to release resources.

If the log file cannot be opened, an error is printed to standard error.

Formatted Logging (`log_output()`) This function allows logging of formatted messages with variable arguments. It uses:

- `va_list, va_start(), va_end()` - Handles variable-length arguments.
- `strcpy(), strcat()` - Constructs the formatted log message before writing it to the log file.

Displaying Logs (`show_logs()`) This function allows users to view all recorded logs by printing the contents of the log file. It operates as follows:

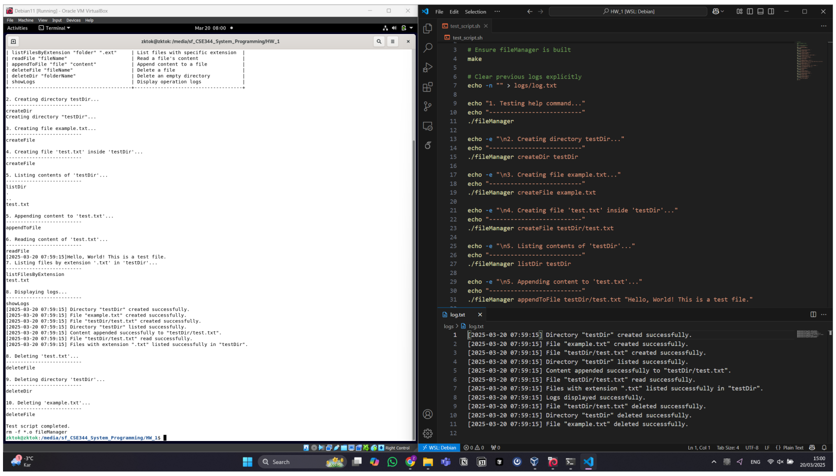
- `fork()` - A child process is created to handle log file reading.
- `open()` - The log file is opened in read mode (`O_RDONLY`).
- `read()` - Reads the log file in chunks and stores the data in a buffer.
- `write()` - Prints the logs to the console.
- `close()` - Closes the log file descriptor after reading.
- `waitpid()` - Ensures that the parent process waits for the child's completion.

If the log file does not exist or an error occurs during reading, an appropriate message is printed.

This module is crucial for tracking all operations and debugging potential issues in the system.

This section provides a detailed explanation of each implemented functionality, including file and directory management operations, logging mechanisms, and process handling.

3 Screenshots



```
| listFilesByExtension "folder" ".ext"    | List files with specific extension  |
| readFile "fileName"                     | Read a file's content               |
| appendToFile "file" "content"           | Append content to a file            |
| deleteFile "fileName"                   | Delete a file                       |
| deleteDir "folderName"                  | Delete an empty directory           |
| showLogs                                | Display operation logs              |
+-----------------------------------------+-------------------------------------+

2. Creating directory testDir...
-------------------------
createDir
Creating directory "testDir"...

3. Creating file example.txt...
-------------------------
createFile

4. Creating file 'test.txt' inside 'testDir'...
-------------------------
createFile

5. Listing contents of 'testDir'...
-------------------------
listDir
.
..
test.txt

5. Appending content to 'test.txt'...
-------------------------
appendToFile

6. Reading content of 'test.txt'...
-------------------------
readFile
[2025-03-20 07:59:15]Hello, World! This is a test file.
7. Listing files by extension '.txt' in 'testDir'...
-------------------------
listFilesByExtension
test.txt

8. Displaying logs...
-------------------------
showLogs
[2025-03-20 07:59:15] Directory "testDir" created successfully.
[2025-03-20 07:59:15] File "example.txt" created successfully.
[2025-03-20 07:59:15] File "testDir/test.txt" created successfully.
[2025-03-20 07:59:15] Directory "testDir" listed successfully.
[2025-03-20 07:59:15] Content appended successfully to "testDir/test.txt".
[2025-03-20 07:59:15] File "testDir/test.txt" read successfully.
[2025-03-20 07:59:15] Files with extension ".txt" listed successfully in "testDir".

8. Deleting 'test.txt'...
-------------------------
deleteFile

9. Deleting directory 'testDir'...
-------------------------
deleteDir

10. Deleting 'example.txt'...
-------------------------
deleteFile

Test script completed.
rm -f *.o fileManager
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$
```

```bash
# Ensure fileManager is built
make

# Clear previous logs explicitly
echo -n "" > logs/log.txt

echo "1. Testing help command..."
echo "-------------------------"
./fileManager

echo -e "\n2. Creating directory testDir..."
echo "-------------------------"
./fileManager createDir testDir

echo -e "\n3. Creating file example.txt..."
echo "-------------------------"
./fileManager createFile example.txt

echo -e "\n4. Creating file 'test.txt' inside 'testDir'..."
echo "-------------------------"
./fileManager createFile testDir/test.txt

echo -e "\n5. Listing contents of 'testDir'..."
echo "-------------------------"
./fileManager listDir testDir

echo -e "\n5. Appending content to 'test.txt'..."
echo "-------------------------"
./fileManager appendToFile testDir/test.txt "Hello, World! This is a test file."
```

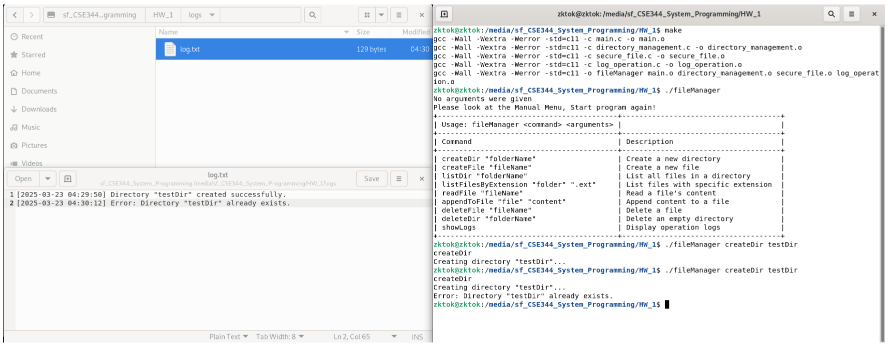
```
[2025-03-20 07:59:15] Directory "testDir" created successfully.
[2025-03-20 07:59:15] File "example.txt" created successfully.
[2025-03-20 07:59:15] File "testDir/test.txt" created successfully.
[2025-03-20 07:59:15] Directory "testDir" listed successfully.
[2025-03-20 07:59:15] Content appended successfully to "testDir/test.txt".
[2025-03-20 07:59:15] File "testDir/test.txt" read successfully.
[2025-03-20 07:59:15] Files with extension ".txt" listed successfully in "testDir".
[2025-03-20 07:59:15] Logs displayed successfully.
[2025-03-20 07:59:15] File "testDir/test.txt" deleted successfully.
[2025-03-20 07:59:15] Directory "testDir" deleted successfully.
[2025-03-20 07:59:15] File "example.txt" deleted successfully.
```

Figure 1: Execution of the test script verifying all operations

All implemented operations were tested using a Bash test script. The test script systematically performed the following steps:

- Created a new directory and verified its existence.
- Created a file and confirmed that the timestamp was correctly written.
- Listed files in the directory and filtered by extension.
- Read the file content and appended new data while ensuring file locking mechanisms worked properly.
- Deleted the file and directory to confirm correct operation.
- Displayed log records to verify that all operations were correctly logged.

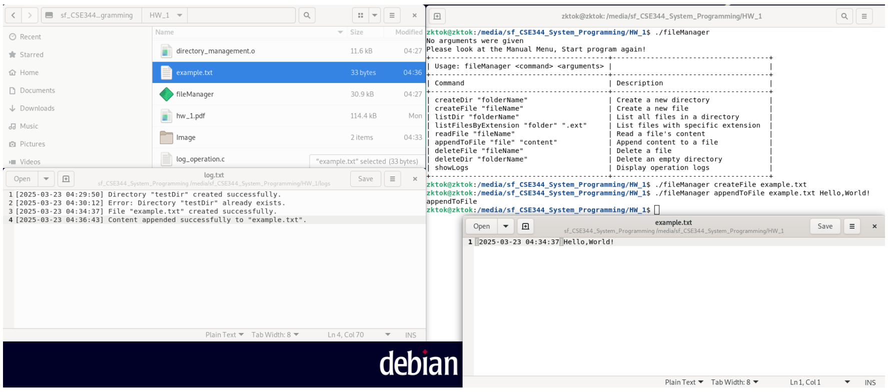
The screenshot in Figure 1 shows the execution of the test script, demonstrating that each function works correctly. The log file was updated successfully for each action, ensuring traceability of all operations. The successful output of the test script confirms the correctness and reliability of the Secure File and Directory Management System.



```
[2025-03-23 04:29:50] Directory "testDir" created successfully.
[2025-03-23 04:30:12] Error: Directory "testDir" already exists.
```

```
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ make
gcc -Wall -Wextra -Werror -std=c11 -c main.c -o main.o
gcc -Wall -Wextra -Werror -std=c11 -c directory_management.c -o directory_management.o
gcc -Wall -Wextra -Werror -std=c11 -c secure_file.c -o secure_file.o
gcc -Wall -Wextra -Werror -std=c11 -c log_operation.c -o log_operation.o
gcc -Wall -Wextra -Werror -std=c11 -o fileManager main.o directory_management.o secure_file.o log_operat
ion.o
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager
No arguments were given
Please look at the Manual Menu, Start program again!
+-------------------------------------------+-------------------------------------+
| Usage: fileManager <command> <arguments> |                                     |
+-------------------------------------------+-------------------------------------+
| Command                                   | Description                         |
+-------------------------------------------+-------------------------------------+
| createDir "folderName"                    | Create a new directory              |
| createFile "fileName"                     | Create a new file                   |
| listDir "folderName"                      | List all files in a directory       |
| listFilesByExtension "folder" ".ext"      | List files with specific extension |
| readFile "fileName"                       | Read a file's content               |
| appendToFile "file" "content"             | Append content to a file            |
| deleteFile "fileName"                     | Delete a file                       |
| deleteDir "folderName"                    | Delete an empty directory           |
| showLogs                                  | Display operation logs              |
+-------------------------------------------+-------------------------------------+
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager createDir testDir
createDir
Creating directory "testDir"...
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager createDir testDir
createDir
Creating directory "testDir"...
Error: Directory "testDir" already exists.
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ █
```

Figure 2: Creating a new directory and verifying its existence

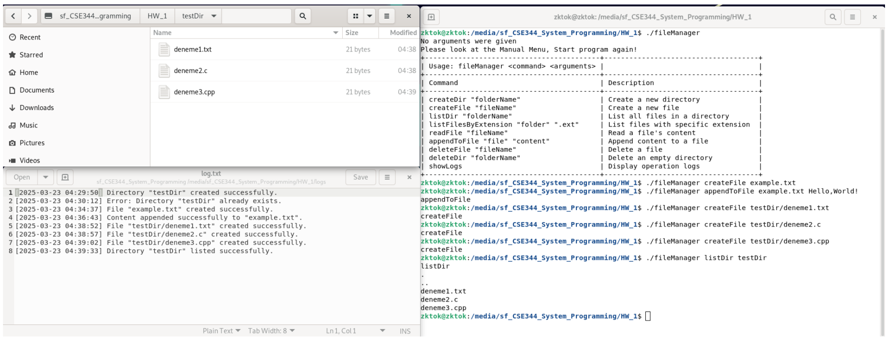


```
[2025-03-23 04:29:50] Directory "testDir" created successfully.
[2025-03-23 04:30:12] Error: Directory "testDir" already exists.
[2025-03-23 04:34:37] File "example.txt" created successfully.
[2025-03-23 04:36:43] Content appended successfully to "example.txt".
```

```
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager
No arguments were given
Please look at the Manual Menu, Start program again!
+-------------------------------------------+-------------------------------------+
| Usage: fileManager <command> <arguments> |                                     |
+-------------------------------------------+-------------------------------------+
| Command                                   | Description                         |
+-------------------------------------------+-------------------------------------+
| createDir "folderName"                    | Create a new directory              |
| createFile "fileName"                     | Create a new file                   |
| listDir "folderName"                      | List all files in a directory       |
| listFilesByExtension "folder" ".ext"      | List files with specific extension |
| readFile "fileName"                       | Read a file's content               |
| appendToFile "file" "content"             | Append content to a file            |
| deleteFile "fileName"                     | Delete a file                       |
| deleteDir "folderName"                    | Delete an empty directory           |
| showLogs                                  | Display operation logs              |
+-------------------------------------------+-------------------------------------+
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager createFile example.txt
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager appendToFile example.txt Hello,World!
appendToFile
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ 
```

```
[2025-03-23 04:34:37]Hello,World!
```

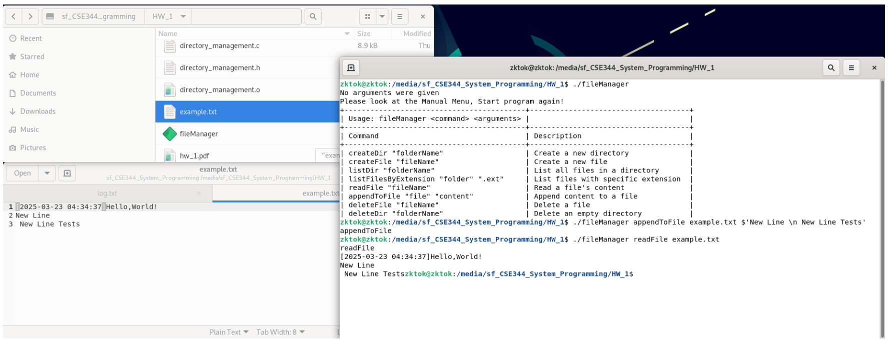
Figure 3: Creating a file and confirming timestamp



```
[2025-03-23 04:29:50] Directory "testDir" created successfully.
[2025-03-23 04:30:12] Error: Directory "testDir" already exists.
[2025-03-23 04:34:37] File "example.txt" created successfully.
[2025-03-23 04:36:43] Content appended successfully to "example.txt".
[2025-03-23 04:38:52] File "testDir/deneme1.txt" created successfully.
[2025-03-23 04:38:57] File "testDir/deneme2.c" created successfully.
[2025-03-23 04:39:02] File "testDir/deneme3.cpp" created successfully.
[2025-03-23 04:39:33] Directory "testDir" listed successfully.
```

```
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager
No arguments were given
Please look at the Manual Menu, Start program again!
+-------------------------------------------+-------------------------------------+
| Usage: fileManager <command> <arguments> |                                     |
+-------------------------------------------+-------------------------------------+
| Command                                   | Description                         |
+-------------------------------------------+-------------------------------------+
| createDir "folderName"                    | Create a new directory              |
| createFile "fileName"                     | Create a new file                   |
| listDir "folderName"                      | List all files in a directory       |
| listFilesByExtension "folder" ".ext"      | List files with specific extension |
| readFile "fileName"                       | Read a file's content               |
| appendToFile "file" "content"             | Append content to a file            |
| deleteFile "fileName"                     | Delete a file                       |
| deleteDir "folderName"                    | Delete an empty directory           |
| showLogs                                  | Display operation logs              |
+-------------------------------------------+-------------------------------------+
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager createFile example.txt
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager appendToFile example.txt Hello,World!
appendToFile
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager createFile testDir/deneme1.txt
createFile
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager createFile testDir/deneme2.c
createFile
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager createFile testDir/deneme3.cpp
createFile
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager listDir testDir
listDir
.
..
deneme1.txt
deneme2.c
deneme3.cpp
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ 
```

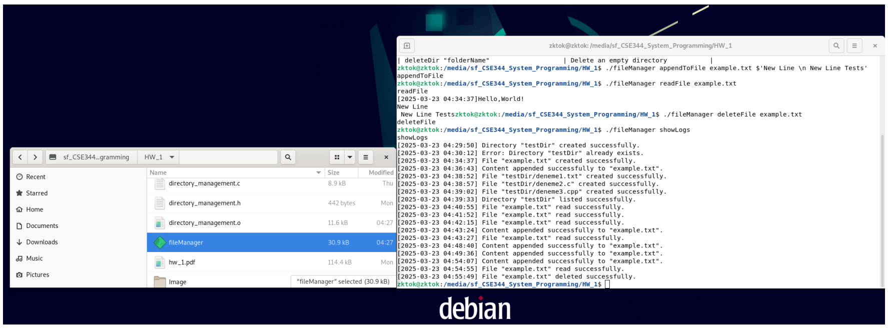
Figure 4: Listing files in the directory



```
[2025-03-23 04:34:37]Hello,World!
New Line
 New Line Tests
```

```
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager
No arguments were given
Please look at the Manual Menu, Start program again!
+-------------------------------------------+----------------------------------------+
| Usage: fileManager <command> <arguments> |                                        |
+-------------------------------------------+----------------------------------------+
| Command                                   | Description                            |
+-------------------------------------------+----------------------------------------+
| createDir "folderName"                    | Create a new directory                 |
| createFile "fileName"                     | Create a new file                      |
| listDir "folderName"                      | List all files in a directory          |
| listFilesByExtension "folder" ".ext"      | List files with specific extension     |
| readFile "fileName"                       | Read a file's content                  |
| appendToFile "file" "content"             | Append content to a file               |
| deleteFile "fileName"                     | Delete a file                          |
| deleteDir "folderName"                    | Delete an empty directory              |
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager appendToFile example.txt $'New Line \n New Line Tests'
appendToFile
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager readFile example.txt
readFile
[2025-03-23 04:34:37]Hello,World!
New Line
 New Line Testszktok@zktok:/media/sf_CSE344_System_Programming/HW_1$
```

Figure 5: Reading and appending content with file locking



```
| deleteDir "folderName"                    | Delete an empty directory              |
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager appendToFile example.txt $'New Line \n New Line Tests'
appendToFile
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager readFile example.txt
readFile
[2025-03-23 04:34:37]Hello,World!
New Line
 New Line Testszktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager deleteFile example.txt
deleteFile
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$ ./fileManager showLogs
showLogs
[2025-03-23 04:29:50] Directory "testDir" created successfully.
[2025-03-23 04:30:12] Error: Directory "testDir" already exists.
[2025-03-23 04:34:37] File "example.txt" created successfully.
[2025-03-23 04:36:43] Content appended successfully to "example.txt".
[2025-03-23 04:38:52] File "testDir/deneme1.txt" created successfully.
[2025-03-23 04:38:57] File "testDir/deneme2.c" created successfully.
[2025-03-23 04:39:02] File "testDir/deneme3.cpp" created successfully.
[2025-03-23 04:39:33] Directory "testDir" listed successfully.
[2025-03-23 04:40:55] File "example.txt" read successfully.
[2025-03-23 04:41:52] File "example.txt" read successfully.
[2025-03-23 04:42:15] File "example.txt" read successfully.
[2025-03-23 04:43:24] Content appended successfully to "example.txt".
[2025-03-23 04:43:27] File "example.txt" read successfully.
[2025-03-23 04:48:40] Content appended successfully to "example.txt".
[2025-03-23 04:49:36] Content appended successfully to "example.txt".
[2025-03-23 04:54:07] Content appended successfully to "example.txt".
[2025-03-23 04:54:55] File "example.txt" read successfully.
[2025-03-23 04:55:49] File "example.txt" deleted successfully.
zktok@zktok:/media/sf_CSE344_System_Programming/HW_1$
```

Figure 6: Deleting file and directory, displaying logs

4 Conclusion

This project successfully implements a Secure File and Directory Management System using Linux system calls, demonstrating key concepts in system programming such as process creation, file handling, directory operations, and logging. The use of `fork()` ensures that each operation is handled by a separate process, allowing efficient execution and preventing blocking issues in the main process. Additionally, file locking mechanisms guarantee data consistency during concurrent write operations.

The logging system plays a crucial role in maintaining traceability by recording all operations performed on files and directories. This feature enhances the system's reliability and allows for easy debugging. The automated testing script verified the correctness of each implemented function, ensuring that all operations function as expected.

Challenges encountered during the development included handling concurrent file access, managing process synchronization, and efficiently implementing system calls without using high-level abstractions. These challenges were overcome by carefully structuring the program logic and utilizing appropriate error-handling mechanisms.

Overall, this project provided valuable insights into system-level programming, deepening the understanding of process control, inter-process communication, and file system management. Future improvements could include adding user authentication for access control, implementing encryption for secure storage, and expanding functionality to support recursive directory operations.