A course project submitted in complete requirements for the award of the degree of

# BACHELOR OF TECHNOLOGY IN

**COMPUTER SCIENCE AND ENGINEERING**

Submitted by

Anirudh Y – 22071A3204

B Akhil Karthik  – 22071A3209

G Sharath – 22071A3214

G Sai Nikhilesh – 22071A3217

Under the guidance of

# Mr

Assistant Professor

Dept. of Computer Science and Engineering



**CERTIFICATE**

This is to certify that have completed their course project work at CSE Department of VNR VJIET, Hyderabad entitled "NEW API FOR FILE HANDLING SYSTEMS" in complete fulfillment of the requirements for the award of B.Tech degree during the academic year 2024-2025. This work is carried out under my supervision and has not been submitted to any other University/Institute for award of any degree/diploma.

**Mr. I Ravindra Kumar Dr. S Nagini**

Assistant Professor Associate Professor and Head

CSE Department CSE Department

VNRVJIET VNRVJIET

## DECLARATION

This is to certify that our project report titled “NEW API FOR FILE HANDLING SYSTEMS **"** submitted to Vallurupalli Nageswara Rao Institute of Engineering and Technology in complete fulfillment of the requirement for the award of Bachelor of Technology in Computer Science and Engineering is a bonafide report to the work carried out by us under the guidance and supervision of **Mr. I Ravindra Kumar**, Assistant Professor, Department of Computer Science and Engineering, Vallurupalli Nageswara Rao Institute of Engineering and Technology. To the best of our knowledge, this has not been submitted in any form to other universities or institutions for the award of any degree or diploma.

## ACKNOWLEDGEMENT

For three and a half years, VNRVJIET has helped us transform ourselves from mere amateurs in the field of Computer Science into skilled engineers capable of handling any given situation in real-time. We are highly indebted to the institute for everything that it has given us. We would like to express our gratitude towards the principal of our institute, **Dr. Challa Dhanunjaya Naidu,** and the Head of the Computer Science & Engineering Department, **Dr. S. Nagini** for their kind cooperation and encouragement who helped us complete the project in the stipulated time. Although we have spent a lot of time and put a lot of effort into this project, it would not have been possible without the motivating support and help of our project guide **Mr.Indurthi Ravindra Kumar** We thank him for his guidance, and constant supervision, and for providing necessary information to complete this project. Our thanks and appreciation also go to all the faculty members, staff members of VNRVJIET, and all our friends who have helped us put this project together.

This project presents a comprehensive File Operations API designed for POSIX-compliant systems. The API encapsulates a wide range of essential file operations, providing a user-friendly interface for developers. Key functionalities include creating, reading, writing, deleting, copying, moving, and renaming files, along with retrieving file information, changing file permissions, and truncating files.

Additionally, the API offers advanced features such as file checksum calculation, keyword search within files, and file locking mechanisms to ensure safe concurrent access. Implemented in C, the API leverages standard libraries and POSIX system calls, ensuring robust error handling and reliability.

By abstracting the complexities of file manipulation, this API streamlines the development process, enabling developers to focus on higher-level application logic. The comprehensive set of functionalities makes it an indispensable tool for any software requiring efficient and consistent file handling capabilities.

**Index**

|  |  |  |
| --- | --- | --- |
| **S No** | **Topic** | **Page No** |
| 1 | Introduction | 8-9 |
| 2 | Literature | 10-11 |
| 3 | Existing System | 12-13 |
| 4 | Proposed System | 14 |
| 5 | Architecture | 15 |
| 6 | Software requirement Specification | 15-16 |
| 7 | Modules | 17 |
| 8 | Implementation | 18-29 |
| 9 | Results | 30-33 |
| 10 | Conclusion and Future Scope | 34 |
| 11 | References | 35 |

**Introduction**

Efficient file handling is a cornerstone of software development, essential for tasks ranging from simple data storage to complex data processing. As software applications grow in complexity and scale, the need for a standardized, robust, and user-friendly interface for file operations becomes paramount. This is especially critical in POSIX-compliant systems, where consistency, reliability, and performance are crucial. To address these needs, we present a comprehensive File Operations API implemented in C. This API offers a streamlined approach to performing a wide array of file manipulation tasks, making it an invaluable tool for developers. By abstracting the complexities of file handling, this API allows developers to focus on higher-level application logic, thereby enhancing productivity and code quality.

The API covers all fundamental file operations such as creating files, reading from files, writing to files, deleting files, copying files, and moving or renaming files. These basic operations are essential for any file manipulation task and provide the foundation for more advanced functionalities. Creating files is as simple as specifying the filename, while reading from and writing to files involve straightforward functions that handle buffer management and data integrity. Deleting files removes them from the filesystem, freeing up resources, and copying files involves duplicating data from one location to another. Moving or renaming files changes their location or name without the need for manual data transfer, thus simplifying file management.

In addition to these basic operations, the API includes features for retrieving file metadata, such as file size and last modification time, which are crucial for managing and auditing file systems. Modifying file permissions to control read, write, and execute rights is another critical feature, especially in multi-user environments where access control is necessary. Truncating files to reduce or extend their size to a specified length allows developers to manage storage space effectively and handle files that need resizing based on their application requirements.

Advanced features of the API further extend its utility. The API can calculate file checksums, which are crucial for verifying file integrity, especially during data transmission or storage. Checksums ensure that the data has not been altered or corrupted. Searching within files to locate specific keywords is an essential feature for text processing and data analysis tasks, enabling applications to quickly find and process relevant information. Moreover, the API includes file locking mechanisms to manage concurrent access to files, ensuring data consistency and preventing conflicts in multi-threaded or multi-process applications. This is particularly important for databases and other applications that require synchronized access to shared resources.

Implemented in C and leveraging POSIX system calls, the API ensures high performance and reliability. POSIX system calls provide a standardized way to perform system-level operations across different UNIX-like operating systems, ensuring compatibility and efficiency. Each function in the API is meticulously designed with thorough error handling, providing meaningful feedback to facilitate debugging and ensure robustness. This comprehensive error handling helps developers quickly identify and resolve issues, reducing downtime and improving the overall stability of applications.

The use of POSIX system calls ensures compatibility and performance across various UNIX-like operating systems. This careful design and implementation make the API a reliable choice for developers working in environments where performance and stability are critical. The API's design principles emphasize simplicity, robustness, and flexibility, allowing it to be easily integrated into various software projects, from small scripts to large-scale enterprise applications.

By using this File Operations API, developers can increase their productivity and the quality of their code. The API abstracts the complexities of file handling, allowing developers to focus on higher-level application logic. This abstraction not only simplifies development but also reduces the potential for errors, leading to more reliable and maintainable software. The API's comprehensive documentation and example programs further aid developers in quickly understanding and utilizing its capabilities.

Whether for simple file manipulation or complex data processing tasks, this API provides the tools necessary for efficient and effective file handling. It offers a standardized and consistent approach to file operations, making it easier for developers to write clean, maintainable, and portable code. As a result, it becomes an indispensable tool for any software project requiring robust and efficient file handling capabilities, making it a valuable addition to the developer's toolkit in various application domains. From managing configuration files in system applications to processing large datasets in scientific research, this File Operations API offers the versatility and reliability needed to handle the diverse challenges of modern software development.

**Literature**

Efficient file handling is a cornerstone of software development, essential for tasks ranging from simple data storage to complex data processing. As software applications grow in complexity and scale, the need for a standardized, robust, and user-friendly interface for file operations becomes paramount. This is especially critical in POSIX-compliant systems, where consistency, reliability, and performance are crucial. To address these needs, we present a comprehensive File Operations API implemented in C. This API offers a streamlined approach to performing a wide array of file manipulation tasks, making it an invaluable tool for developers. By abstracting the complexities of file handling, this API allows developers to focus on higher-level application logic, thereby enhancing productivity and code quality.

The API covers all fundamental file operations such as creating files, reading from files, writing to files, deleting files, copying files, and moving or renaming files. These basic operations are essential for any file manipulation task and provide the foundation for more advanced functionalities. Creating files is as simple as specifying the filename, while reading from and writing to files involve straightforward functions that handle buffer management and data integrity. Deleting files removes them from the filesystem, freeing up resources, and copying files involves duplicating data from one location to another. Moving or renaming files changes their location or name without the need for manual data transfer, thus simplifying file management.

In addition to these basic operations, the API includes features for retrieving file metadata, such as file size and last modification time, which are crucial for managing and auditing file systems. Modifying file permissions to control read, write, and execute rights is another critical feature, especially in multi-user environments where access control is necessary. Truncating files to reduce or extend their size to a specified length allows developers to manage storage space effectively and handle files that need resizing based on their application requirements.

Advanced features of the API further extend its utility. The API can calculate file checksums, which are crucial for verifying file integrity, especially during data transmission or storage. Checksums ensure that the data has not been altered or corrupted. Searching within files to locate specific keywords is an essential feature for text processing and data analysis tasks, enabling applications to quickly find and process relevant information. Moreover, the API includes file locking mechanisms to manage concurrent access to files, ensuring data consistency and preventing conflicts in multi-threaded or multi-process applications. This is particularly important for databases and other applications that require synchronized access to shared resources.

Implemented in C and leveraging POSIX system calls, the API ensures high performance and reliability. POSIX system calls provide a standardized way to perform system-level operations across different UNIX-like operating systems, ensuring compatibility and efficiency. Each function in the API is meticulously designed with thorough error handling, providing meaningful feedback to facilitate debugging and ensure robustness. This comprehensive error handling helps developers quickly identify and resolve issues, reducing downtime and improving the overall stability of applications.

The use of POSIX system calls ensures compatibility and performance across various UNIX-like operating systems. This careful design and implementation make the API a reliable choice for developers working in environments where performance and stability are critical. The API's design principles emphasize simplicity, robustness, and flexibility, allowing it to be easily integrated into various software projects, from small scripts to large-scale enterprise applications.

By using this File Operations API, developers can increase their productivity and the quality of their code. The API abstracts the complexities of file handling, allowing developers to focus on higher-level application logic. This abstraction not only simplifies development but also reduces the potential for errors, leading to more reliable and maintainable software. The API's comprehensive documentation and example programs further aid developers in quickly understanding and utilizing its capabilities.

Whether for simple file manipulation or complex data processing tasks, this API provides the tools necessary for efficient and effective file handling. It offers a standardized and consistent approach to file operations, making it easier for developers to write clean, maintainable, and portable code. As a result, it becomes an indispensable tool for any software project requiring robust and efficient file handling capabilities, making it a valuable addition to the developer's toolkit in various application domains. From managing configuration files in system applications to processing large datasets in scientific research, this File Operations API offers the versatility and reliability needed to handle the diverse challenges of modern software development.

**Existing Systems**

## Standard C Library (stdio.h)

* **Low-Level Operations**: Provides basic functions without high-level abstractions, making complex tasks harder.
* **Minimal Error Handling**: Limited error handling, requiring manual intervention for robustness.
* **Concurrency Issues**: Lacks built-in support for safe concurrent file access.

## POSIX System Calls

* **Complexity**: Requires deep understanding of system-level programming.
* **Portability Issues**: Not all systems fully comply with POSIX, leading to potential cross-platform issues.
* **Manual Resource Management**: Requires manual handling of file descriptors, increasing the risk of resource leaks.

## Higher-Level Libraries (e.g., Boost.Filesystem, Java File class)

* **Overhead**: Additional abstraction layers can impact performance.
* **Dependency Management**: Adds complexity and increases the size of the final executable.
* **Platform Dependencies**: May have platform-specific behaviors, reducing code portability.

**Proposed System**

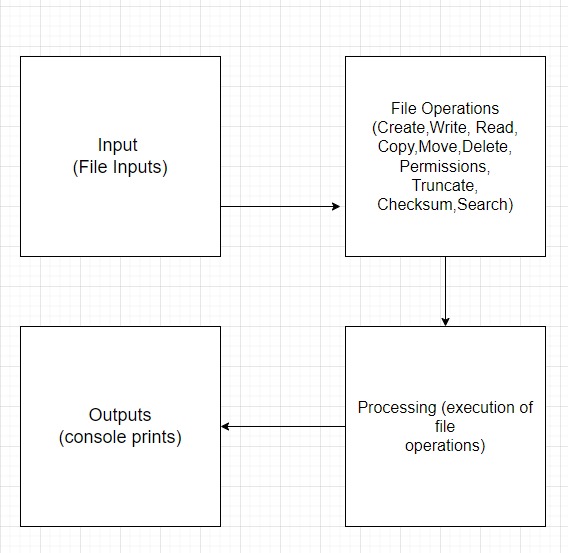
The POSIX File Operations API is designed to provide a comprehensive and user-friendly interface for managing file operations on POSIX-compliant systems. It integrates basic and advanced file handling functionalities, offering a robust and efficient solution for developers.

### Advantages

1. **Comprehensive Functionality**:
   * Combines essential file operations (create, read, write, delete, copy, move/rename) with advanced features (checksum calculation, keyword search, file locking) in a single API.
2. **Ease of Use**:
   * High-level abstractions simplify complex file handling tasks, reducing the amount of code developers need to write and maintain.
3. **Robust Error Handling**:
   * Built-in error handling mechanisms provide meaningful feedback and enhance the reliability of applications.
4. **Concurrency Management**:
   * Integrated file locking mechanisms ensure safe concurrent access, preventing data corruption and race conditions.
5. **Portability**:
   * POSIX-compliant design ensures compatibility across various operating systems, making the API suitable for cross-platform applications.

By addressing the shortcomings of existing systems, the proposed POSIX File Operations API aims to streamline file operations, improve code quality, and enhance developer productivity.

**Architecture**

**Software requirement Specifications**

## Hardware Requirements

* **RAM**: 16GB
* **Processor**: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
* **Storage**: 512 GB

## Software Requirements

* **Ubuntu**:
  + Ubuntu is both a philosophy emphasizing interconnectedness and a popular Linux-based operating system known for its user-friendly interface and open-source commitment. The term "Ubuntu" means "humanity towards others" in Zulu.
* **C Compiler (gcc)**:
  + A C compiler translates human-readable C code (.c files) into machine code executable by computers. On some systems, "cc" is a symbolic link to a specific compiler like GCC (GNU Compiler Collection), which handles preprocessing, compiling, and linking stages of compilation.

## Functional Requirements

* **File Operations API**:
  + Comprehensive functionality for basic and advanced file operations including create, read, write, delete, copy, move/rename, change permissions, and truncate files.
* **Locking System**:
  + Protects shared resources in multiprocessor systems by serializing access to prevent conflicts and ensure safe access among multiple processors.
* **Shared Data Management**:
  + Prevents data inconsistency when multiple processors access the same data simultaneously by implementing protocols or locking schemes.
* **Cache Coherence**:
  + Maintains a coherent view of data stored in multiple local caches to resolve conflicts arising from changes in cached copies without proper notification.

# Non functional Requirements:

**Reliability**:

* The API should have minimal downtime and be resilient to failures, ensuring that file operations are not lost or corrupted.

**Scalability**:

* The API should accommodate growth in the number of file operations and users without significant degradation in performance or functionality.

**Interoperability**:

* The API should integrate smoothly with other software or systems, facilitating seamless data exchange without loss of information.

**Performance**:

* The API should provide quick response times for file operations, ensuring a responsive user experience.

**Compatibility**:

* The API should be compatible with different devices, operating systems, and development environments to ensure broad usability.

**Compliance**:

* The API should comply with relevant laws, regulations, and industry standards concerning data protection, privacy, and accessibility.

**Modules**

**File Operations:** The API provides comprehensive functions for opening, closing, reading, writing, and seeking within files, accommodating both synchronous and asynchronous modes. These functions form the core of file manipulation, allowing developers to perform essential tasks efficiently. The create\_file, read\_file, and write\_file functions handle the basic operations needed to create a new file, read its contents into a buffer, and append new data to it. The close function ensures that files are properly closed after operations, freeing system resources. The seek function allows developers to navigate to specific positions within a file, which is particularly useful for handling large files or processing specific sections of a file. Synchronous modes ensure that operations are completed before moving on to the next task, guaranteeing data integrity. Asynchronous modes, on the other hand, allow for non-blocking operations, improving performance in applications that require concurrent file access.

**File Management:** Managing files efficiently is crucial for maintaining an organized filesystem and ensuring data integrity. The API includes functions for creating, deleting, copying, moving, and renaming files, with robust error handling and support for large files. The delete\_file function safely removes files from the system, ensuring that no residual data is left behind. The copy\_file function duplicates the contents of a file to a new location, while the move\_file function renames or relocates files within the filesystem. These operations are optimized for handling large files, ensuring that even extensive data can be managed effectively. Enhanced error handling mechanisms provide detailed feedback in case of failures, allowing developers to diagnose and resolve issues quickly. By automating these file management tasks, the API simplifies the development process and reduces the potential for human error.

**Permissions and Security:** Ensuring secure access to files is paramount in multi-user environments and applications that handle sensitive data. The API offers functions to set and retrieve file permissions, providing a robust security framework. The change\_file\_permissions function allows developers to specify the access rights for different users, controlling who can read, write, or execute a file. This ensures that only authorized personnel can modify or access sensitive files. The get\_file\_info function retrieves the current permissions and other metadata, providing a comprehensive overview of a file's security settings. By enforcing strict access controls, the API helps protect data integrity and confidentiality, making it suitable for applications that require stringent security measures.

**File Analysis:** Understanding the properties and characteristics of files is essential for effective file management and data processing. The API includes functions to retrieve and analyze file metadata, such as size, type, and modification dates. The get\_file\_info function provides detailed information about a file, including its size, which is useful for managing storage space and optimizing application performance. The get\_last\_modification\_time function retrieves the last modification date, allowing developers to track changes and implement version control. Additionally, the API can analyze file contents to calculate checksums, ensuring data integrity by detecting any alterations or corruption. These analytical capabilities provide developers with the insights needed to manage files effectively, optimize storage solutions, and ensure data reliability.

Overall, the File Operations API offers a comprehensive suite of tools for efficient file handling, management, security, and analysis. By abstracting complex file operations into simple, high-level functions, it empowers developers to build robust and efficient applications with ease. This API not only enhances productivity and code quality but also ensures that applications can handle file operations reliably and securely across various POSIX-compliant systems.

**Implementation**

**Program:**

**Header file : “file\_operations.h”**

#ifndef FILE\_OPERATIONS\_H

#define FILE\_OPERATIONS\_H

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/stat.h>

#include <unistd.h>

#include <fcntl.h>

#include <time.h>

// Basic File Operations

int create\_file(const char \*filename);

int read\_file(const char \*filename, char \*buffer, size\_t size);

int write\_file(const char \*filename, const char \*data);

int delete\_file(const char \*filename);

int copy\_file(const char \*source, const char \*destination);

int move\_file(const char \*source, const char \*destination);

int get\_file\_info(const char \*filename);

int file\_exists(const char \*filename);

time\_t get\_last\_modification\_time(const char \*filename);

int change\_file\_permissions(const char \*filename, mode\_t mode);

int truncate\_file(const char \*filename, off\_t length);

unsigned long calculate\_file\_checksum(const char \*filename);

int search\_within\_file(const char \*filename, const char \*keyword);

#endif // FILE\_OPERATIONS\_H

**C source file : “file\_operations.c” (for implementations of the functions in header file)**

#include "file\_operations.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/stat.h>

#include <unistd.h>

#include <fcntl.h>

#include <time.h>

#include <sys/file.h>

// Basic operations implementation

int create\_file(const char \*filename) {

FILE \*file = fopen(filename, "w");

if (!file) {

return -1; // Error creating file

}

fclose(file);

return 0; // Success

}

int read\_file(const char \*filename, char \*buffer, size\_t size) {

FILE \*file = fopen(filename, "r");

if (!file) {

return -1; // Error opening file

}

fread(buffer, 1, size, file);

fclose(file);

return 0; // Success

}

int write\_file(const char \*filename, const char \*data) {

FILE \*file = fopen(filename, "a");

if (!file) {

return -1; // Error opening file

}

fwrite(data, 1, strlen(data), file);

fclose(file);

return 0; // Success

}

int delete\_file(const char \*filename) {

if (remove(filename) != 0) {

return -1; // Error deleting file

}

return 0; // Success

}

int copy\_file(const char \*source, const char \*destination) {

char buffer[1024];

size\_t bytes;

FILE \*src = fopen(source, "r");

if (!src) {

return -1; // Error opening source file

}

FILE \*dest = fopen(destination, "w");

if (!dest) {

fclose(src);

return -1; // Error opening destination file

}

while ((bytes = fread(buffer, 1, sizeof(buffer), src)) > 0) {

fwrite(buffer, 1, bytes, dest);

}

fclose(src);

fclose(dest);

return 0; // Success

}

int move\_file(const char \*source, const char \*destination) {

if (rename(source, destination) != 0) {

return -1; // Error moving/renaming file

}

return 0; // Success

}

int get\_file\_info(const char \*filename) {

struct stat st;

if (stat(filename, &st) != 0) {

return -1; // Error getting file info

}

return (int)st.st\_size; // Return file size

}

int file\_exists(const char \*filename) {

struct stat buffer;

return (stat(filename, &buffer) == 0);

}

time\_t get\_last\_modification\_time(const char \*filename) {

struct stat attrib;

if (stat(filename, &attrib) != 0) {

return (time\_t)-1; // Error getting file attributes

}

return attrib.st\_mtime;

}

// New Basic Operations

int change\_file\_permissions(const char \*filename, mode\_t mode) {

if (chmod(filename, mode) != 0) {

return -1; // Error changing file permissions

}

return 0; // Success

}

int truncate\_file(const char \*filename, off\_t length) {

if (truncate(filename, length) != 0) {

return -1; // Error truncating file

}

return 0; // Success

}

// Advanced Operations

unsigned long calculate\_file\_checksum(const char \*filename) {

FILE \*file = fopen(filename, "rb");

if (!file) {

return 0; // Error opening file

}

unsigned long checksum = 0;

int c;

while ((c = fgetc(file)) != EOF) {

checksum += c;

}

fclose(file);

return checksum;

}

int search\_within\_file(const char \*filename, const char \*keyword) {

FILE \*file = fopen(filename, "r");

if (!file) {

return -1; // Error opening file

}

char line[1024];

int line\_num = 0;

int found = 0;

while (fgets(line, sizeof(line), file)) {

line\_num++;

if (strstr(line, keyword)) {

printf("Found '%s' in %s at line %d: %s", keyword, filename, line\_num, line);

found = 1;

}

}

fclose(file);

return found ? 0 : -1; // Success if found, otherwise error

}

**C Source file : “main.c” (to write the driver program)**

#include <stdio.h>

#include <time.h>

#include "file\_operations.h"

int main()

{

// Demonstrating create\_file

const char \*file1 = "file1.txt";

printf("Creating file: %s\n", file1);

create\_file(file1);

printf("File created: %s\n\n", file1);

// Demonstrating write\_file

const char \*file2 = "file2.txt";

const char \*data2 = "This is file2 data.\n";

printf("Creating file: %s\n", file2);

create\_file(file2);

write\_file(file2, data2);

printf("Data written to %s: %s\n", file2, data2);

// Demonstrating read\_file

const char \*file3 = "file3.txt";

const char \*data3 = "This is file3 data.\n";

printf("Creating file: %s\n", file3);

create\_file(file3);

write\_file(file3, data3);

char buffer3[256];

read\_file(file3, buffer3, sizeof(buffer3));

printf("Content of %s: %s\n", file3, buffer3);

// Demonstrating file\_exists

printf("Checking file existence for %s...\n", file3);

if (file\_exists(file3))

{

printf("%s exists\n", file3);

}

else

{

printf("%s does not exist\n", file3);

}

// Demonstrating get\_file\_info

printf("\nGetting file size of %s...\n", file3);

printf("Size of %s: %d bytes\n", file3, get\_file\_info(file3));

// Demonstrating get\_last\_modification\_time

printf("\nGetting last modification time of %s...\n", file3);

time\_t mod\_time = get\_last\_modification\_time(file3);

printf("Last modification time of %s: %s", file3, ctime(&mod\_time));

// Demonstrating copy\_file

const char \*file4 = "file4.txt";

printf("\nCopying %s to %s...\n", file3, file4);

if (copy\_file(file3, file4) == 0){

printf("%s copied to %s\n", file3, file4);

}

read\_file(file4, buffer3, sizeof(buffer3));

printf("Content of %s: %s\n", file4, buffer3);

// Demonstrating move\_file

const char \*file5 = "file5.txt";

printf("\nMoving %s to %s...\n", file4, file5);

if (move\_file(file4, file5) == 0)

{

printf("%s moved to %s\n", file4, file5);

}

read\_file(file5, buffer3, sizeof(buffer3));

printf("Content of %s: %s\n", file5, buffer3);

// Demonstrating delete\_file

printf("\nDeleting %s...\n", file2);

if (delete\_file(file2) == 0)

{

printf("%s deleted\n", file2);

}

if (file\_exists(file2))

{

printf("%s exists\n", file2);

}

else

{printf("%s does not exist\n", file2);

}

// Demonstrating change\_file\_permissions and write\_file

printf("\nChanging file permissions of %s to read-only...\n", file1);

if (change\_file\_permissions(file1, S\_IRUSR | S\_IRGRP | S\_IROTH) == 0)

{

printf("Permissions changed successfully\n");

}

else

{

printf("Error changing file permissions\n");

}

// Attempt to write to read-only file

const char \*data1 = "Trying to write to read-only file.\n";

printf("Attempting to write to %s after setting to read-only:\n", file1);

if (write\_file(file1, data1) != 0)

{

printf("Error: Unable to write to read-only file %s\n", file1);

} // Demonstrating truncate\_file

const char \*file6 = "file6.txt";

const char \*data6 = "This is a test data to create a file with more than 300 bytes. "

"Lorem ipsum dolor sit amet, consectetur adipiscing elit. "

"Phasellus imperdiet, nulla et dictum interdum, nisi lorem egestas odio, "

"vitae scelerisque enim ligula venenatis dolor. "

"Maecenas nisl est, ultrices nec congue eget, auctor vitae massa.\n";

printf("Creating and writing to file: %s\n", file6);

create\_file(file6);

write\_file(file6, data6); printf("Size of %s before truncation: %d bytes\n", file6, get\_file\_info(file6));

printf("\nTruncating %s to 150 bytes...\n", file6);

if (truncate\_file(file6, 150) == 0)

{

printf("File truncated successfully\n");

}

else

{

printf("Error truncating file\n");

}

printf("Size of %s after truncation: %d bytes\n", file6, get\_file\_info(file6));

// Demonstrating calculate\_file\_checksum

printf("\nCalculating checksum of %s...\n", file5);

unsigned long checksum = calculate\_file\_checksum(file5);

printf("Checksum of %s: %lu\n", file5, checksum);

// Demonstrating search\_within\_file

printf("\nSearching for 'data' in %s...\n", file5);

if (search\_within\_file(file5, "data") == 0)

{

printf("Keyword found\n");

}

else

{

printf("Keyword not found\n");

}

return 0;

}

**Results:**

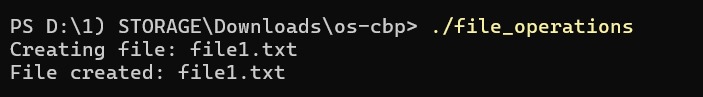
**int create\_file(const char \*filename):**

Description: Creates a new file with the given filename.

System Calls: Uses fopen() to create the file and fclose() to close it.

Libraries: <stdio.h> for fopen() and fclose().

**Output:**



**int read\_file(const char \*filename, char \*buffer, size\_t size):**

Description: Reads data from a file into a buffer.

System Calls: Uses fopen(), fread(), and fclose().

Libraries: <stdio.h> for fopen(), fread(), and fclose().

**Output:**



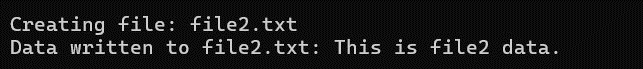
**int write\_file(const char \*filename, const char \*data):**

Description: Appends data to a file.

System Calls: Uses fopen(), fwrite(), and fclose().

Libraries: <stdio.h> for fopen(), fwrite(), and fclose().

**output:**



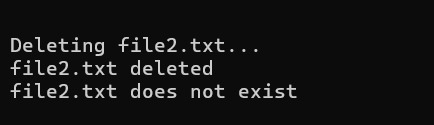
**int delete\_file(const char \*filename):**

Description: Deletes a file.

System Calls: Uses remove().

Libraries: <stdio.h> for remove().

**Output:**



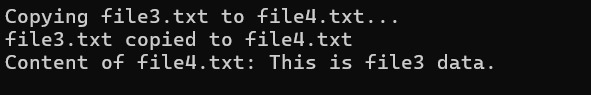
**int copy\_file(const char \*source, const char \*destination):**

Description: Copies a file from source to destination.

System Calls: Uses fopen(), fread(), fwrite(), and fclose().

Libraries: <stdio.h> for fopen(), fread(), fwrite(), and fclose().

**Output:**



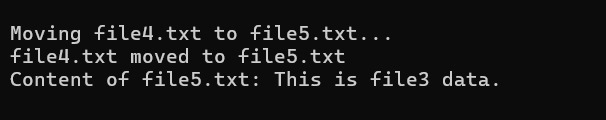
**int move\_file(const char \*source, const char \*destination):**

Description: Moves or renames a file.

System Calls: Uses rename().

Libraries: <stdio.h> for rename().

**Output:**



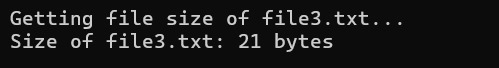
**int get\_file\_info(const char \*filename):**

Description: Retrieves the file size.

System Calls: Uses stat() to get file information.

Libraries: <sys/stat.h> for stat().

**Output:**



**int file\_exists(const char \*filename):**

Description: Checks if a file exists.

System Calls: Uses stat().

Libraries: <sys/stat.h> for stat().

**Output:**



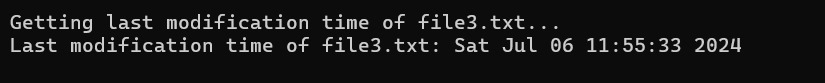
**time\_t get\_last\_modification\_time(const char \*filename):**

Description: Gets the last modification time of a file.

System Calls: Uses stat().

Libraries: <sys/stat.h> for stat().

**Output:**



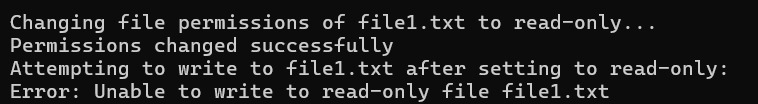
**int change\_file\_permissions(const char \*filename, mode\_t mode):**

Description: Changes the permissions of a file.

System Calls: Uses chmod().

Libraries: <sys/stat.h> for chmod().

**Output:**



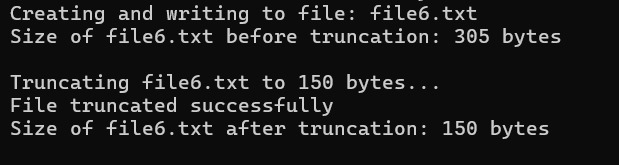
**int truncate\_file(const char \*filename, off\_t length):**

Description: Truncates a file to a specified length.

System Calls: Uses truncate().

Libraries: <unistd.h> for truncate().

**Output:**



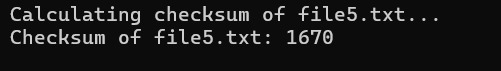
**unsigned long calculate\_file\_checksum(const char \*filename):**

Description: Calculates the checksum of a file.

System Calls: Uses fopen(), fgetc(), and fclose().

Libraries: <stdio.h> for fopen(), fgetc(), and fclose().

**Output:**



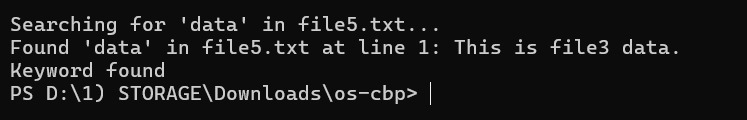
**int search\_within\_file(const char \*filename, const char \*keyword):**

Description: Searches for a keyword within a file and prints lines containing the keyword.

System Calls: Uses fopen(), fgets(), fclose().

Libraries: <stdio.h> for fopen(), fgets(), and fclose(), <string.h> for strstr().

**Output:**



**Conclusion and Future Scope**

In conclusion, the development of a comprehensive File Operations API presents a structured approach to managing file-related tasks within software systems. Through the implementation of fundamental operations such as file creation, reading, writing, deletion, copying, moving, and more, the API aims to enhance efficiency, reliability, and scalability in handling file operations.

The architecture of the API emphasizes modularity and extensibility, enabling seamless integration with existing systems and compatibility across different platforms. Key components like error handling, concurrency management, compatibility layers, and performance optimization further enhance the robustness and effectiveness of the API.

### Future Scope

Looking forward, there are several avenues for future enhancement and expansion of the File Operations API:

* **Enhanced Error Handling**: Implementing more sophisticated error handling mechanisms to provide detailed feedback and improve fault tolerance.
* **Advanced Concurrency Management**: Integrating advanced concurrency control techniques to better manage simultaneous file accesses and improve system performance under high load.
* **Integration with Cloud Services**: Extending the API to support seamless interaction with cloud storage solutions, leveraging APIs provided by major cloud providers.
* **Security Enhancements**: Strengthening file security measures, such as encryption and access control, to protect sensitive data from unauthorized access.
* **Performance Optimization**: Continuously refining file operations algorithms and data structures to achieve optimal performance across diverse hardware and software environments.

**References**

* <https://www.programiz.com/c-programming/c-file-input-output>
* <https://www.w3schools.com/c/c_files.php>
* https://www.geeksforgeeks.org/basics-file-handling-c/
* <https://www.cprogramming.com/tutorial/cfileio.html>
* <https://www.javatpoint.com/file-handling-in-c>
* <https://stackoverflow.com/questions/1653958/how-to-create-and-use-custom-header-> files-in-c