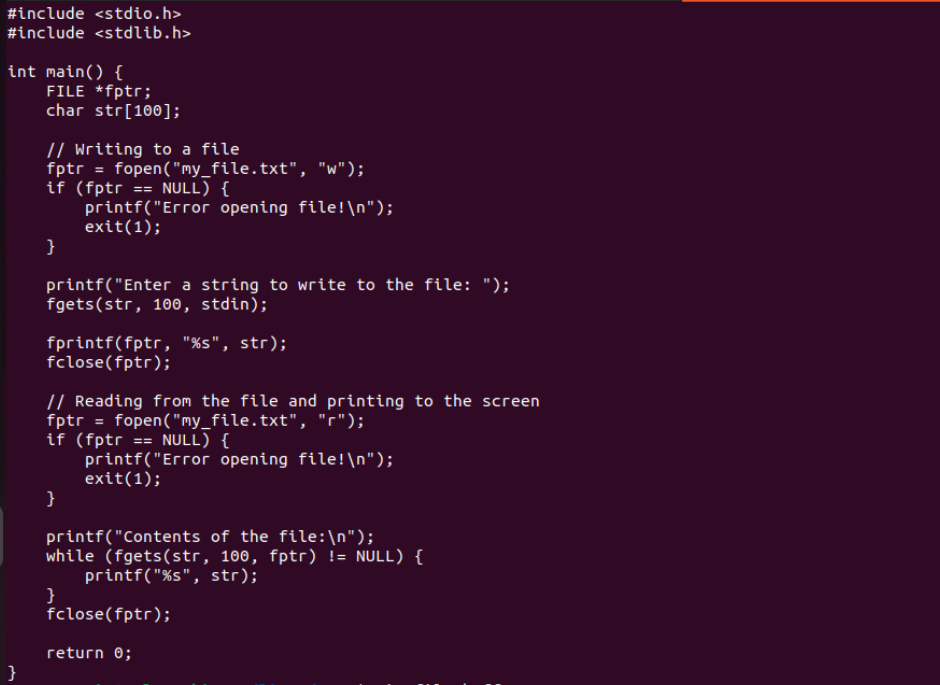
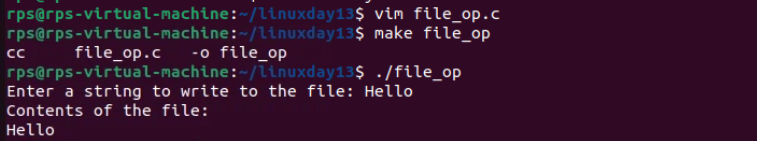
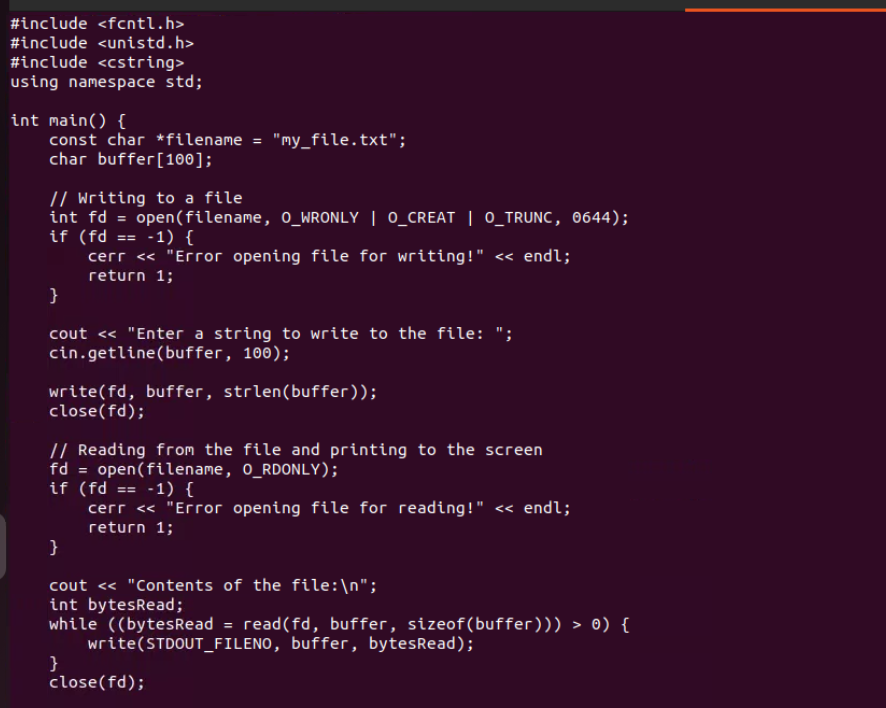
**Date:01-08-2024**

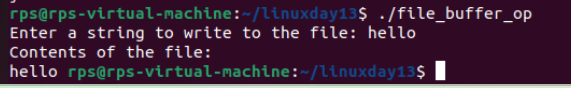
**Day\_13\_linux\_Assignments:**

1.

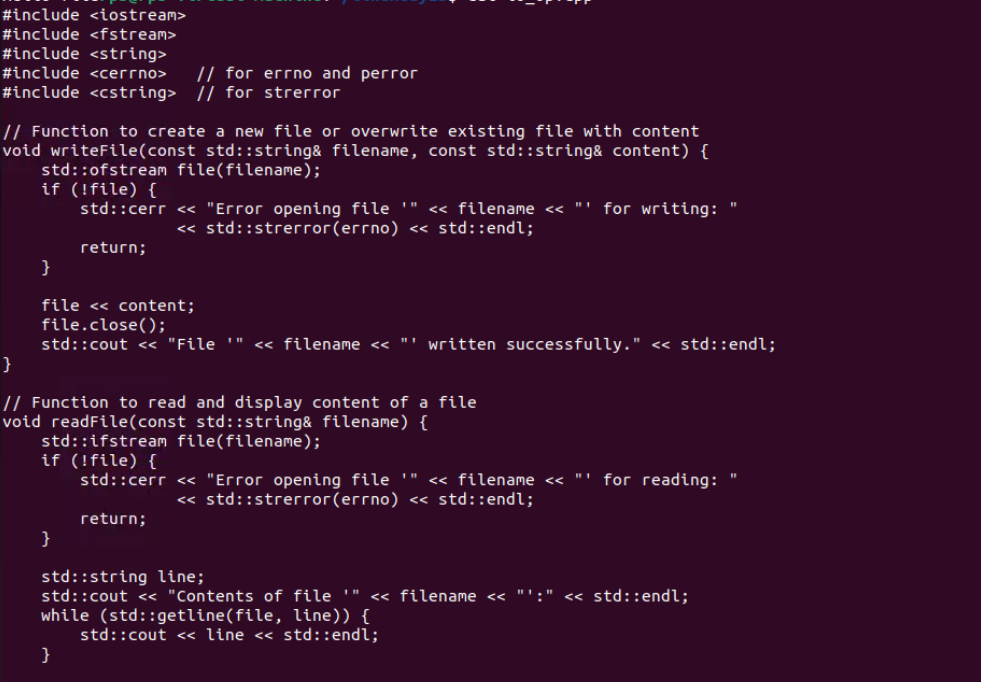


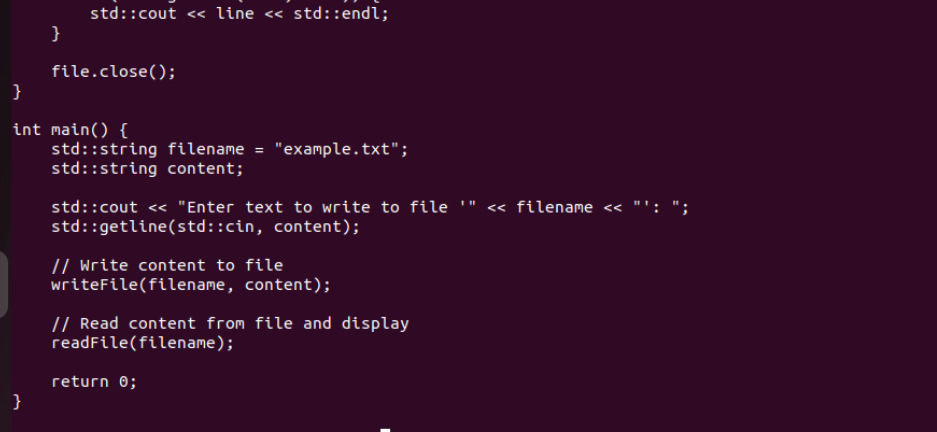




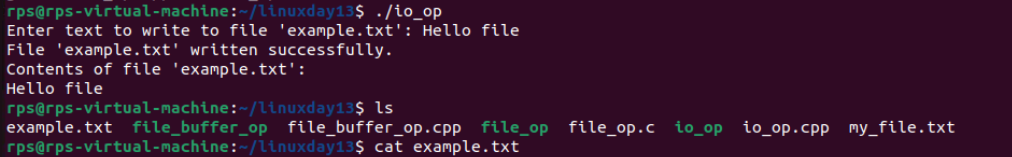


3. Develop a C++ application that utilizes system calls to perform basic file I/O operations.Specific Requirements:Create a new file if it doesn't exist.Write user-provided text content to the file.Read the contents of the file and display them on the console.Implement robust error handling for file operations.





Output:



4.The open system call is a fundamental function in Unix-like operating systems used to open a file or device for reading, writing, or appending. It returns a file descriptor, a non-negative integer used to reference the open file in subsequent system calls.

#include <fcntl.h>

int open(const char pathname, int flags, ... / mode\_t mode \*/);

Gemini was just updated.  See update

Conversation with Gemini

Write a c code use open close read and write to write onto file and print on screen using same

Show drafts

C

#include <stdio.h>

#include <stdlib.h>

int main() {

FILE \*fptr;

char str[100];

// Writing to a file

fptr = fopen("my\_file.txt", "w");

if (fptr == NULL) {

printf("Error opening file!\n");

exit(1);

}

printf("Enter a string to write to the file: ");

fgets(str, 100, stdin);

fprintf(fptr, "%s", str);

fclose(fptr);

// Reading from the file and printing to the screen

fptr = fopen("my\_file.txt", "r");

if (fptr == NULL) {

printf("Error opening file!\n");

exit(1);

}

printf("Contents of the file:\n");

while (fgets(str, 100, fptr) != NULL) {

printf("%s", str);

}

fclose(fptr);

return 0;

}

Use code with caution.

This code performs the following steps:

Includes necessary headers: stdio.h for standard input/output functions and stdlib.h for the exit function.

Declares a file pointer: FILE \*fptr to store the file pointer.

Declares a character array: char str[100] to store the string.

Opens the file in write mode: fptr = fopen("my\_file.txt", "w");

Checks for file opening error: If the file cannot be opened, prints an error message and exits the program.

Prompts the user to enter a string: printf("Enter a string to write to the file: ");

Reads the string from the user: fgets(str, 100, stdin);

Writes the string to the file: fprintf(fptr, "%s", str);

Closes the file: fclose(fptr);

Opens the file in read mode: fptr = fopen("my\_file.txt", "r");

Checks for file opening error: Similar to the previous check.

Prints a message indicating the contents of the file: printf("Contents of the file:\n");

Reads the file line by line: while (fgets(str, 100, fptr) != NULL) { ... }

Prints each line to the screen: printf("%s", str);

Closes the file: fclose(fptr);

Returns 0 to indicate successful program execution.

This code effectively demonstrates how to open, write to, close, reopen, read from, and print the contents of a file using the fopen, fclose, fprintf, and fgets functions in C.

Write above in cpp using system call

Show drafts

Understanding the Task:

We're aiming to recreate the functionality of the previous C code using system calls in C++. This involves:

Creating a file

Writing data to the file

Reading data from the file

Printing the contents to the screen

Key System Calls:

open: Creates or opens a file

write: Writes data to an open file

read: Reads data from an open file

close: Closes an open file

Code Implementation:

C++

#include <iostream>

#include <fcntl.h>

#include <unistd.h>

using namespace std;

int main() {

const char \*filename = "my\_file.txt";

char buffer[100];

// Writing to a file

int fd = open(filename, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (fd == -1) {

cerr << "Error opening file for writing!" << endl;

return 1;

}

cout << "Enter a string to write to the file: ";

cin.getline(buffer, 100);

write(fd, buffer, strlen(buffer));

close(fd);

// Reading from the file and printing to the screen

fd = open(filename, O\_RDONLY);

if (fd == -1) {

cerr << "Error opening file for reading!" << endl;

return 1;

}

cout << "Contents of the file:\n";

int bytesRead;

while ((bytesRead = read(fd, buffer, sizeof(buffer))) > 0) {

write(STDOUT\_FILENO, buffer, bytesRead);

}

close(fd);

return 0;

}

Use code with caution.

Explanation:

Include necessary headers: iostream for input/output, fcntl.h for file control flags, and unistd.h for system calls.

Declare variables: filename for the file name, buffer for storing data, and fd for the file descriptor.

Open the file for writing: open with O\_WRONLY, O\_CREAT, and O\_TRUNC flags to create if not exists, truncate if exists, and open for writing.

Error handling: Check if fd is -1, indicating an error.

Get user input: Read a string from the user using cin.getline.

Write to the file: Use write to write the string to the file.

Close the file: Use close to close the file.

Open the file for reading: open with O\_RDONLY flag.

Error handling: Check for errors as before.

Read from the file and print: Use read to read data into the buffer, and write to print it to the screen (using STDOUT\_FILENO).

Close the file: Close the file.

Key Points:

System calls provide a lower-level interface to file operations.

Error handling is crucial when using system calls.

open flags control file creation and access modes.

read and write operate on file descriptors and byte counts.

STDOUT\_FILENO represents the standard output file descriptor.

This code demonstrates the use of system calls to perform file operations in C++, providing a more direct interaction with the operating system.

Note: While this code uses system calls, it's generally recommended to use higher-level abstractions like fstream for most file operations in C++ for better readability and portability.

Would you like to explore using fstream for this task?

Create a problem statement on above to write code

Show drafts

Problem Statement

Develop a C++ application that utilizes system calls to perform basic file I/O operations.

Specific Requirements:

Create a new file if it doesn't exist.

Write user-provided text content to the file.

Read the contents of the file and display them on the console.

Implement robust error handling for file operations.

Additional Considerations:

Explore the efficiency and performance implications of using system calls compared to standard C++ file I/O libraries.

Consider potential security vulnerabilities associated with direct system call usage.

By effectively addressing these requirements, the application should demonstrate a solid understanding of file handling concepts and system call interactions.

Write full details about system api open

Show drafts

System API: open

Overview

The open system call is a fundamental function in Unix-like operating systems used to open a file or device for reading, writing, or appending. It returns a file descriptor, a non-negative integer used to reference the open file in subsequent system calls.

Syntax

C

#include <fcntl.h>

int open(const char pathname, int flags, ... / mode\_t mode \*/);

Use code with caution.

Parameters

pathname: A null-terminated character string specifying the path to the file or device to be opened.

flags: An integer specifying the flags that control how the file is opened.

mode: An optional mode\_t argument used only if the file is created (when the O\_CREAT flag is set). It specifies the permissions for the new file.

Flags

The flags argument is a bitwise OR of one or more of the following flags:

O\_RDONLY: Open for reading only.

O\_WRONLY: Open for writing only.

O\_RDWR: Open for both reading and writing.

O\_CREAT: Create the file if it does not exist.

O\_EXCL: Used with O\_CREAT, fails if the file already exists.

O\_TRUNC: Truncate the file to zero length if it exists.

O\_APPEND: Append writes to the end of the file.

O\_NONBLOCK: Non-blocking open (for devices).

O\_SYNC: Synchronize file writes (slower).

O\_DSYNC: Synchronize data writes (faster than O\_SYNC).

Open ( ):

The **open**() system call opens the file specified by *pathname*.

Pathname- A pointer to a string that specifies the file to be opened. This can be either an absolute or a relative path.

Flags- There are three access mode in flags a) for read only we use O\_RDONLY

b) for write only we use O\_WRONLY c) for both we use O\_RDWR.

If we want to create file then we use O\_CREAT and when we use it then we need more .More is used to specifies the permissions to use in case a new file is created.It used some file permission like S\_IRUSR- which is used for read permisssion.S\_IWUSR-used for write permission

Close ():

closes a file descriptor, so that it no longer refers to any file and may be reused.

Read () :

attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

**fd**: The file descriptor from which to read data. This is an integer value obtained from a previous call to open, socket, or other functions that return a file descriptor.

**buf**: A pointer to a buffer where the read data will be stored.

**count**: The maximum number of bytes to read. This specifies the size of the buffer.

\*\*\* File Descriptor- In Linux and other Unix-like operating systems, a **file descriptor** is a low-level integer identifier used to access files or other input/output (I/O) resources such as sockets, pipes, or terminals.

0 is used for standard input.(Stdin)

1 is used for standaed output.(stdout)

2 is used for standard error(Stderr)

Write():

**write**():

writes up to *count* bytes from the buffer starting at

buf to the file referred to by the file descriptor *fd*.

**fd**: The file descriptor to which data will be written. This is an integer value obtained from a previous call to open,

socket, or other functions that return a file descriptor.

**buf**: A pointer to the buffer containing the data to be written. This is typically an array of bytes.

**count**: The number of bytes to write from the buffer.

Socekt():

creates an endpoint for communication and returns a file

descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

**domain**: Specifies the communication domain (e.g., AF\_INET for IPv4, AF\_INET6 for IPv6).

**type**: Specifies the socket type (e.g., SOCK\_STREAM for TCP, SOCK\_DGRAM for UDP).

**protocol**: Specifies a particular protocol (usually set to 0 to choose the default for the type).

Bind:

It is associated with socket with local address and port.

**sockfd**: The socket file descriptor.

**addr**: A pointer to a sockaddr structure that specifies the address and port.

**addrlen**: The size of the address structure.

Connect:

It is used to connect a socket to a remote address.

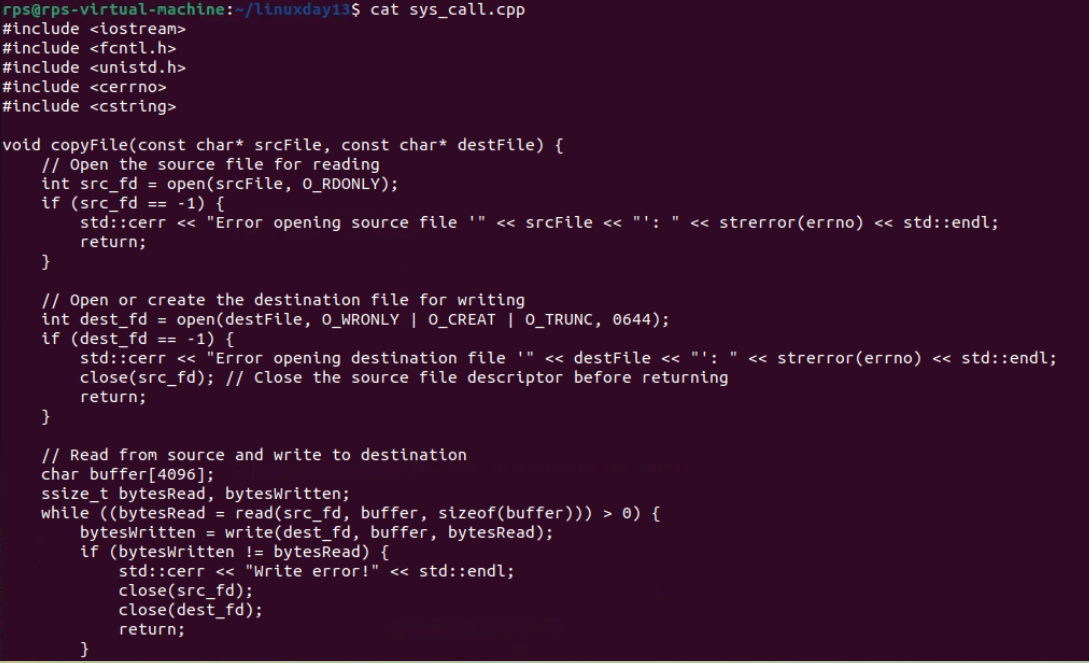
**sockfd**: The socket file descriptor.

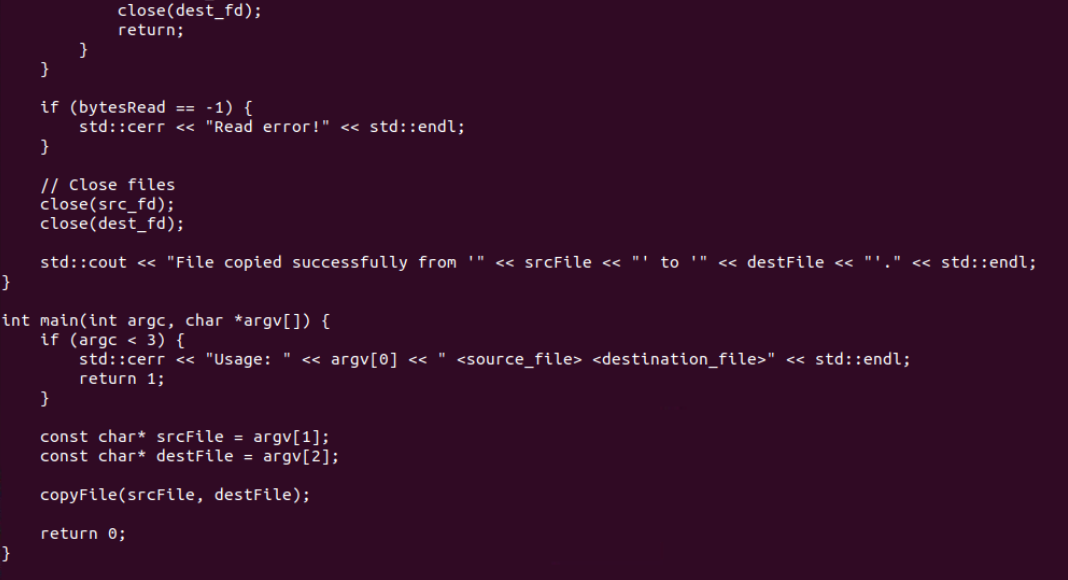
**addr**: A pointer to a sockaddr structure that specifies the remote address and port.

**addrlen**: The size of the address structure.

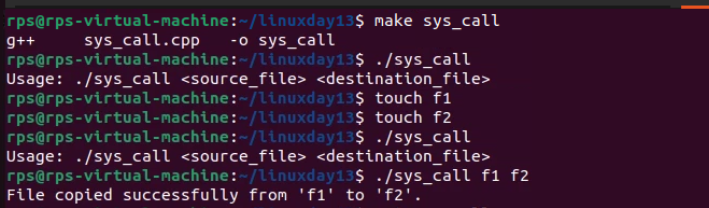
5. File I/O and Manipulation:

Copy a File: Write a C++ program that takes two file paths as command-line arguments. Use open, read, write, and close system calls to copy the contents of the source file to the destination file. Handle potential errors (e.g., file not found, permission denied).

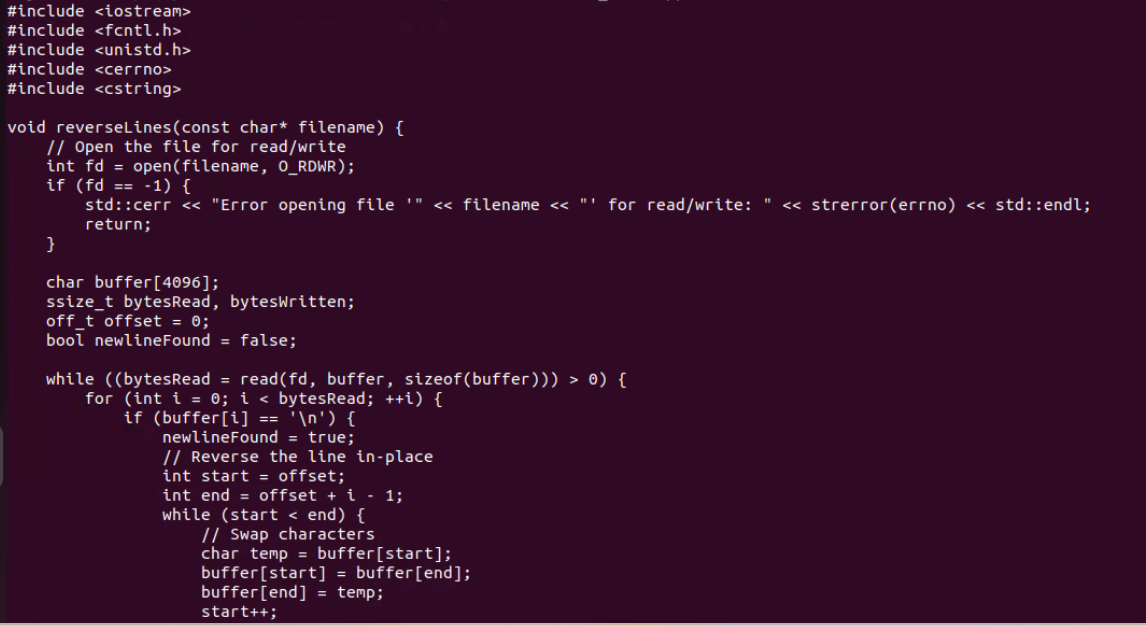


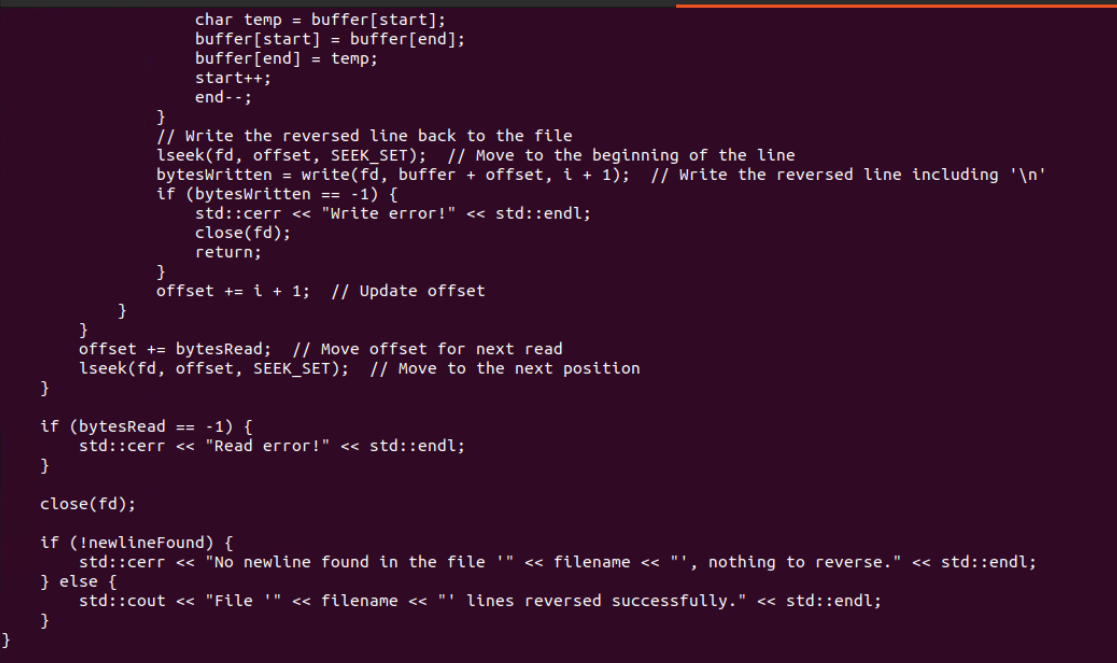


Output:



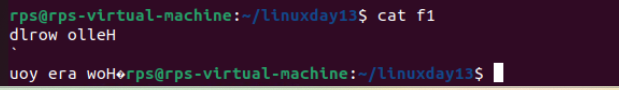
6. Reverse a File: Write a C++ program that reads the contents of a file line by line, reverses each line in-place, and then writes the reversed lines back to the same file. Use system calls like open, read, write, lseek, and close to achieve this.



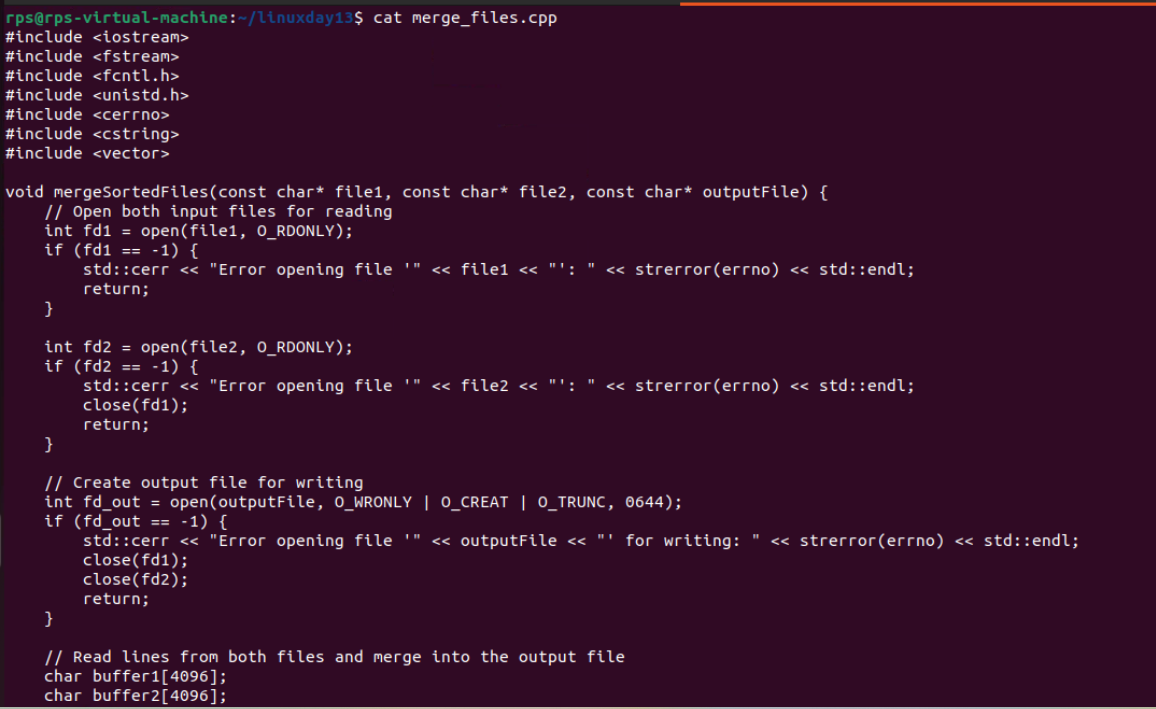


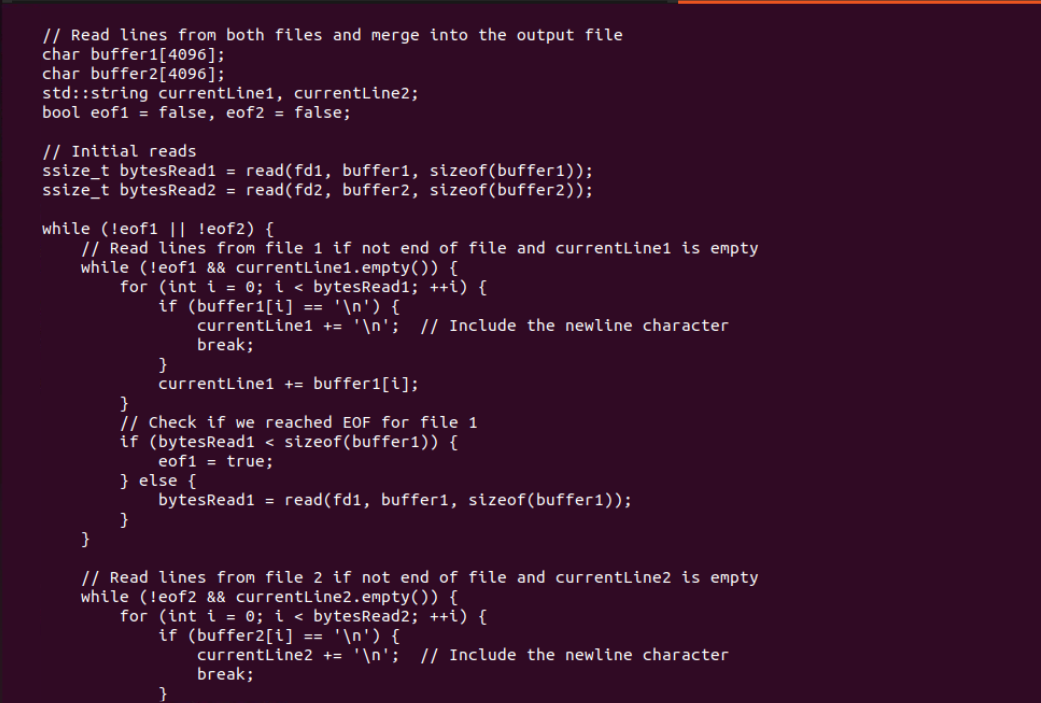


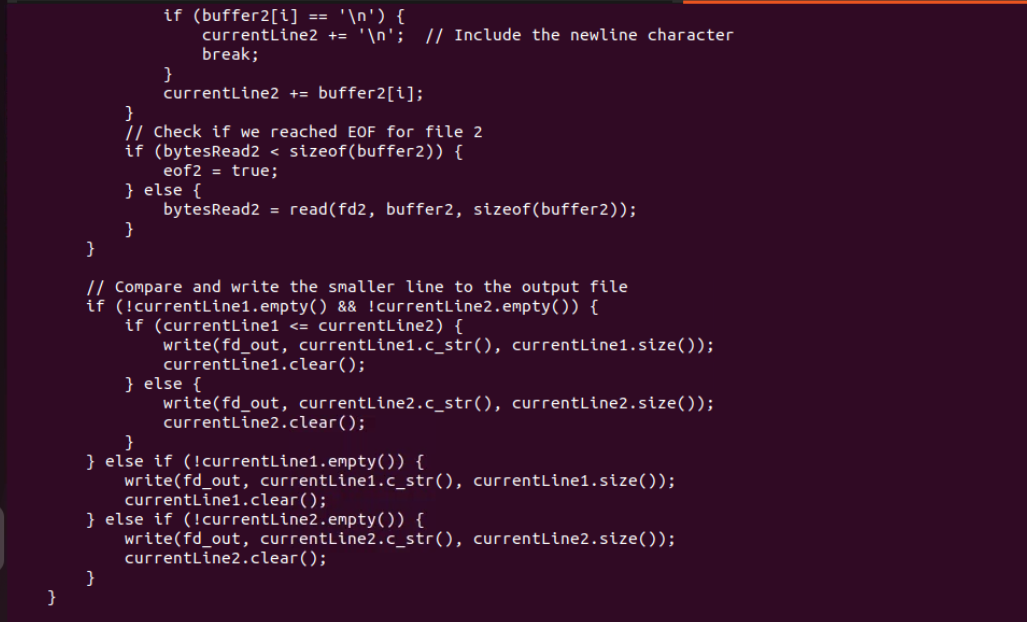
Output:

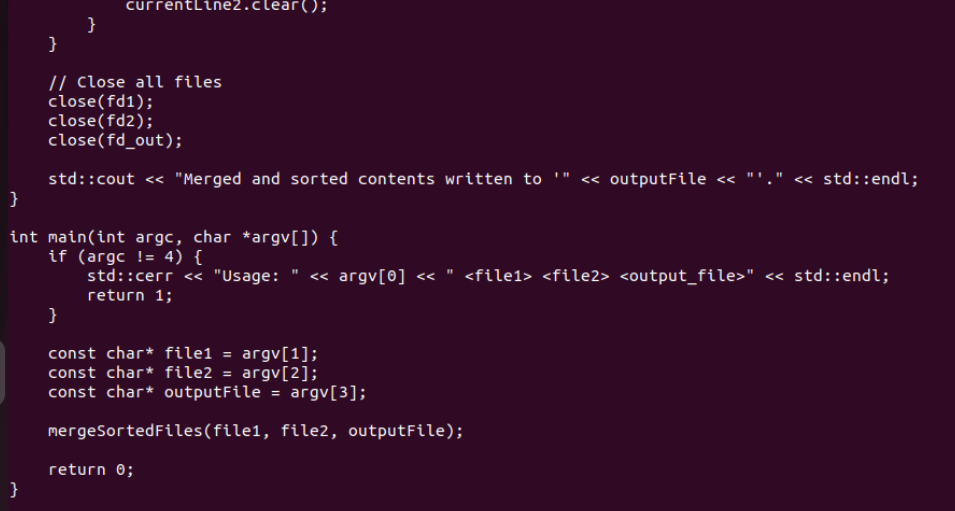


7. Merge Two Sorted Files: Write a C++ program that takes two sorted text files as input and creates a new file containing the merged and sorted contents. Use appropriate system calls for file handling and consider memory efficiency when handling large files.

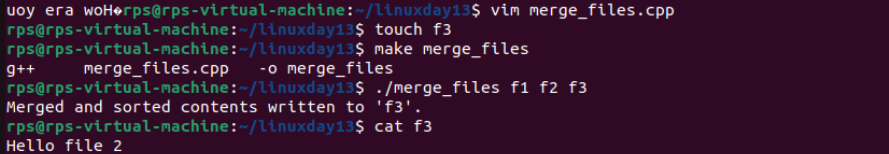




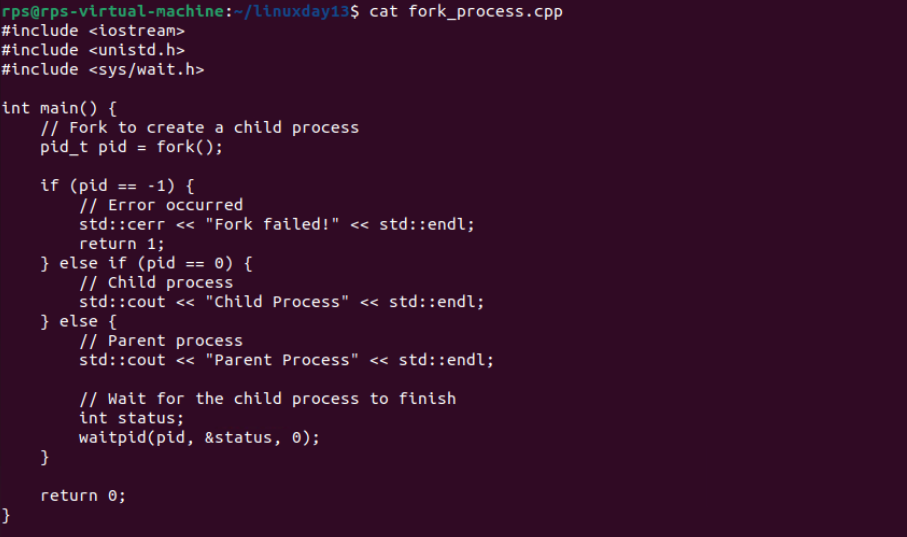


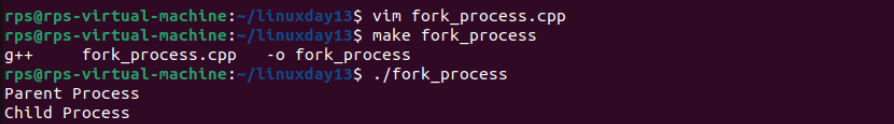


Output:

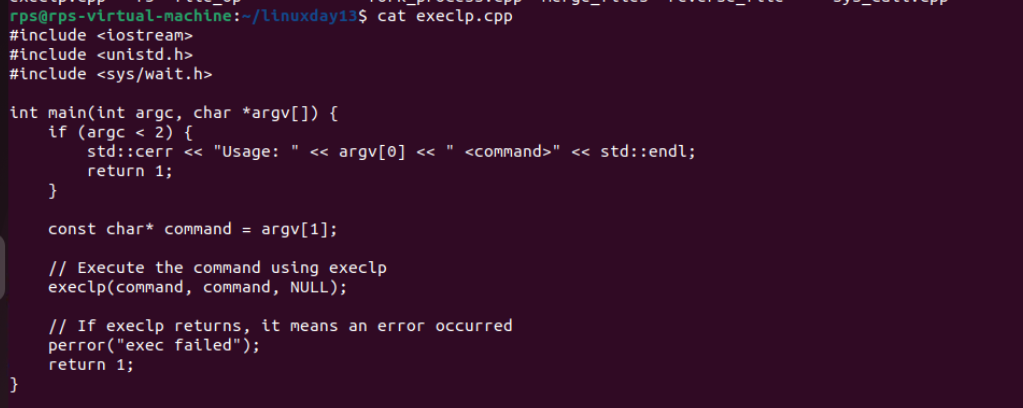


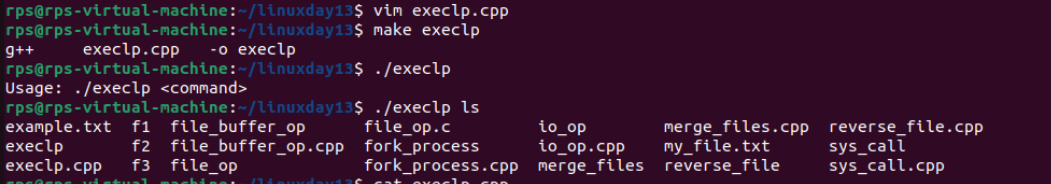
8. Process Control and Inter-Process Communication:Create a Child Process with fork: Write a C++ program that uses fork to create a child process. The parent process should print "Parent Process", and the child process should print "Child Process". Use wait or similar system calls to ensure the parent waits for the child to finish before exiting.



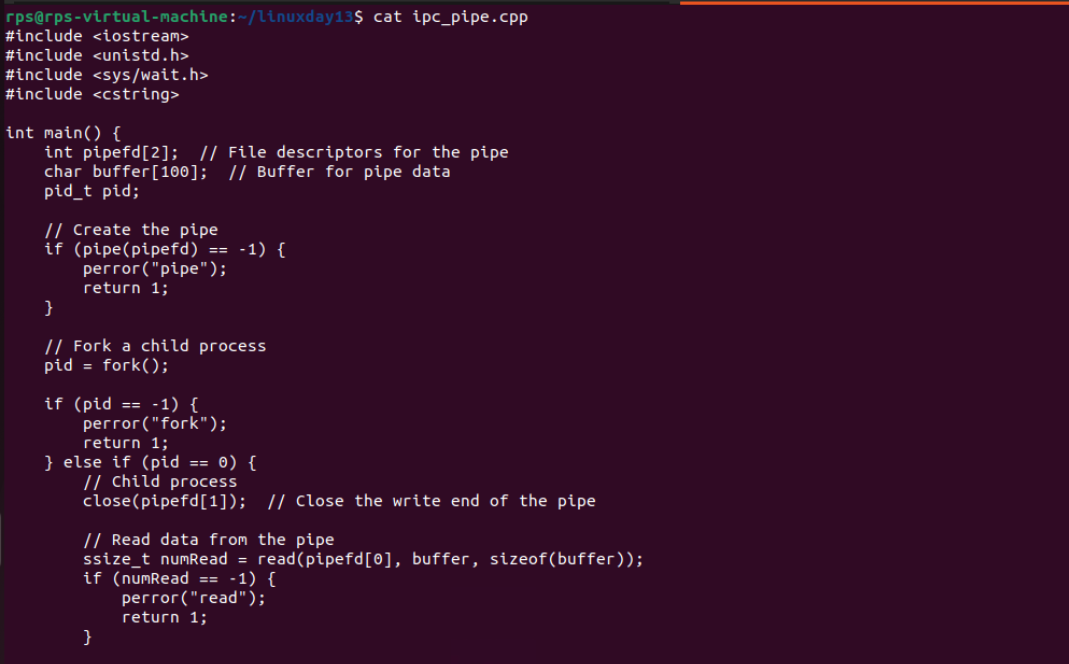


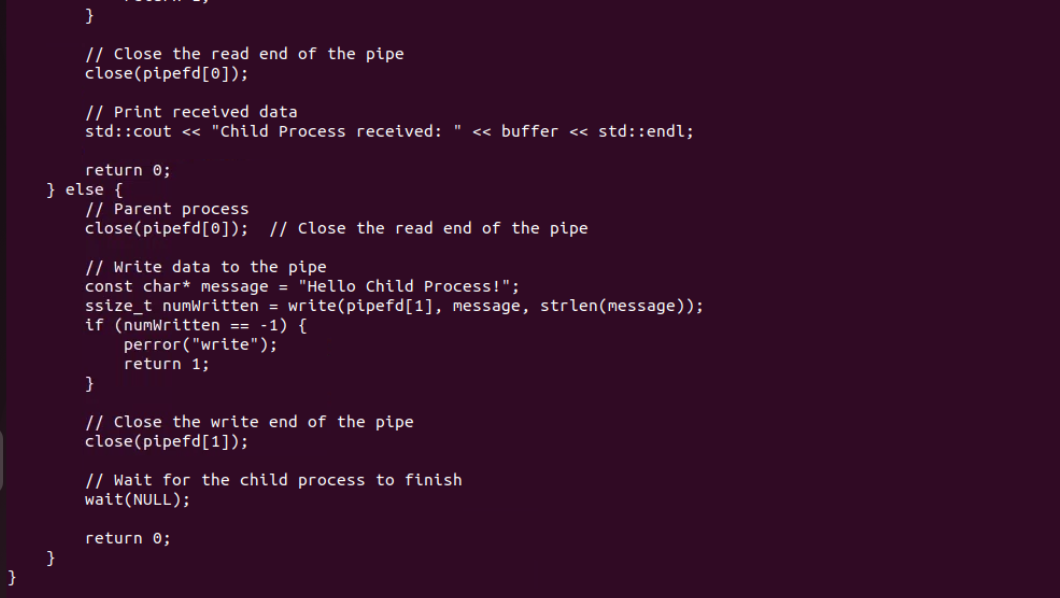
9. Execute a Shell Command: Write a C++ program that takes a shell command as a string argument and uses exec system calls (e.g., execlp or execv) to execute that command. Handle errors if the command execution fails.



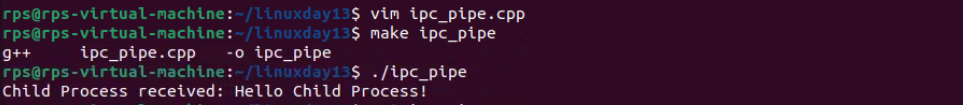


10. Inter-Process Communication with Pipes: Write a C++ program that demonstrates inter-process communication using pipes. One process should write data to a pipe, and another process should read from the pipe and print the received data. Leverage pipe and fork system calls.



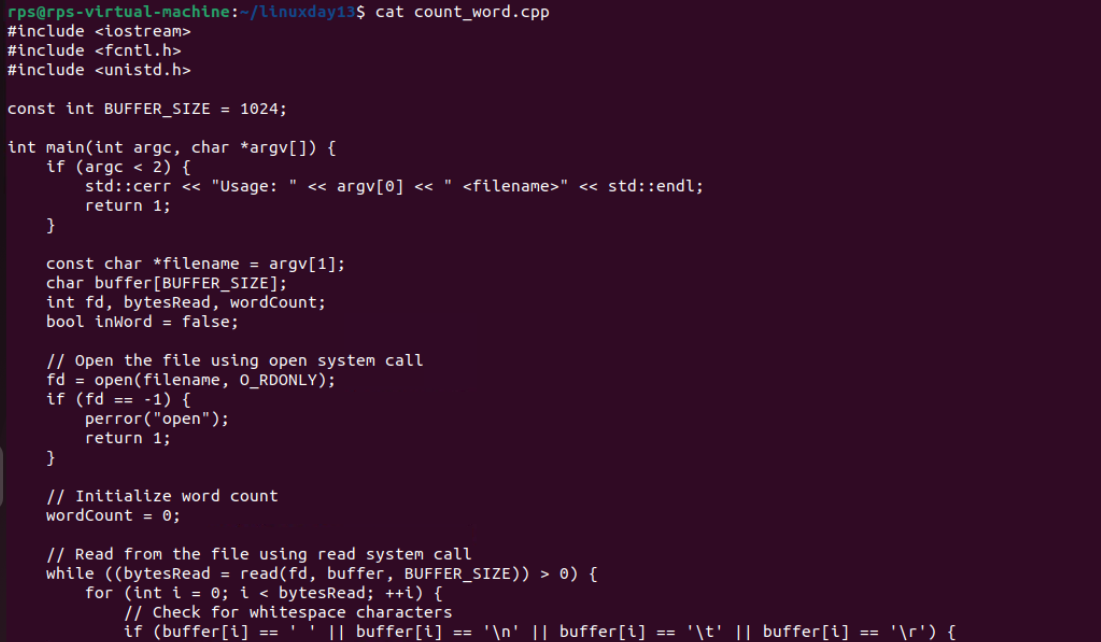


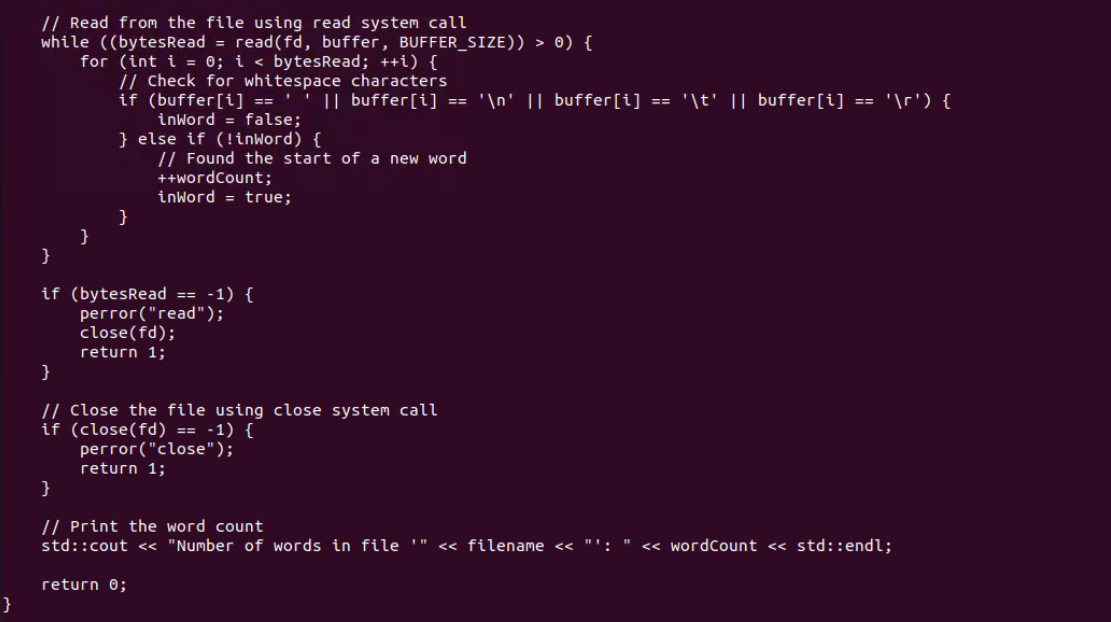
Output:



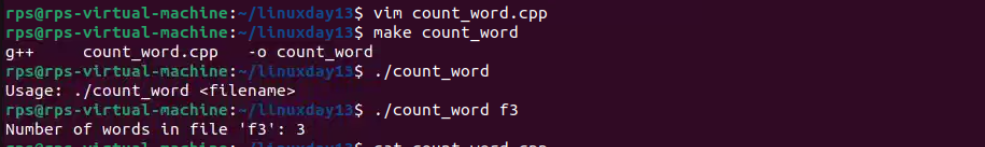
11. Text Processing and System Information:

Count Words in a File: Write a C++ program that reads a text file and counts the number of words in it. Use open, read, and close system calls to access the file. Be mindful of delimiters and whitespace characters when counting words.

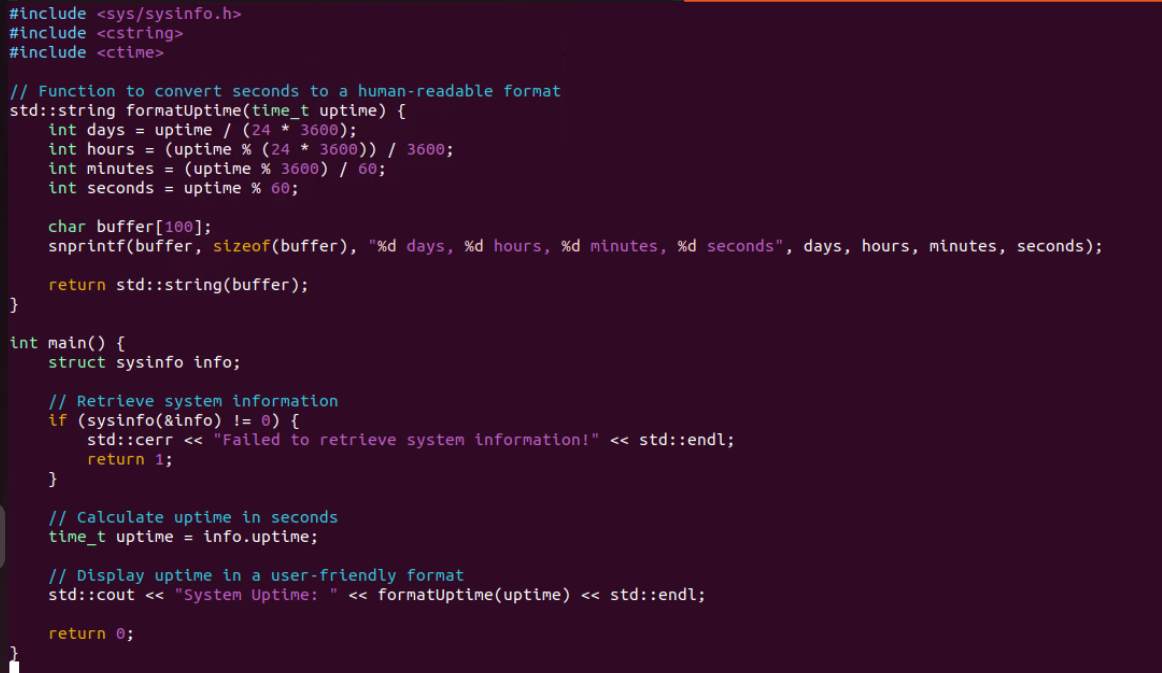




Output:



12. Get System Uptime: Write a C++ program that retrieves the system's uptime (time since it was last booted) using appropriate system calls (e.g., getuptime on Linux). Display the uptime information in a user-friendly format.

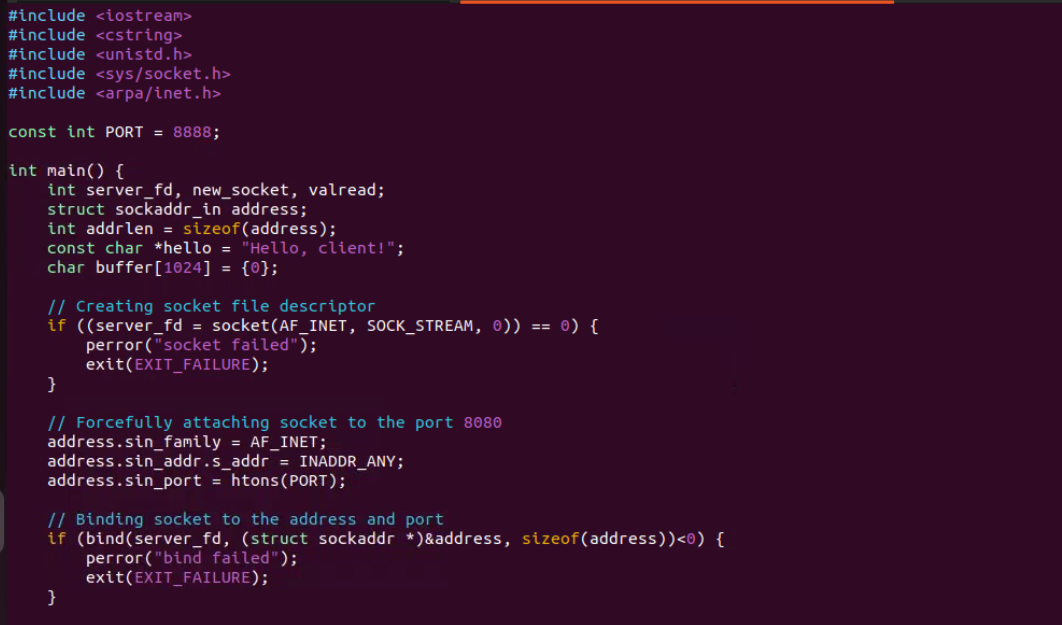


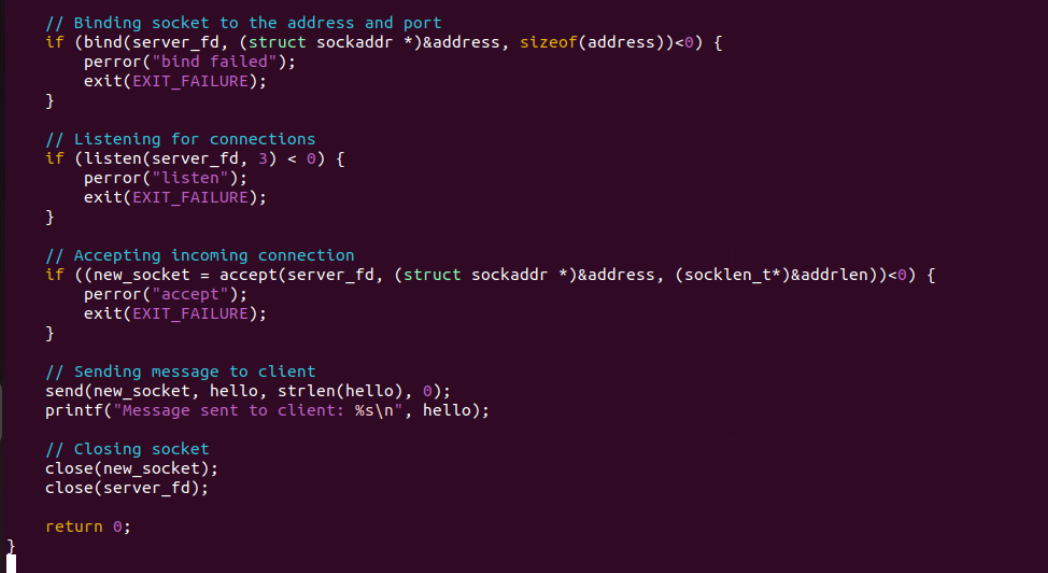


13. Network Programming (Bonus):

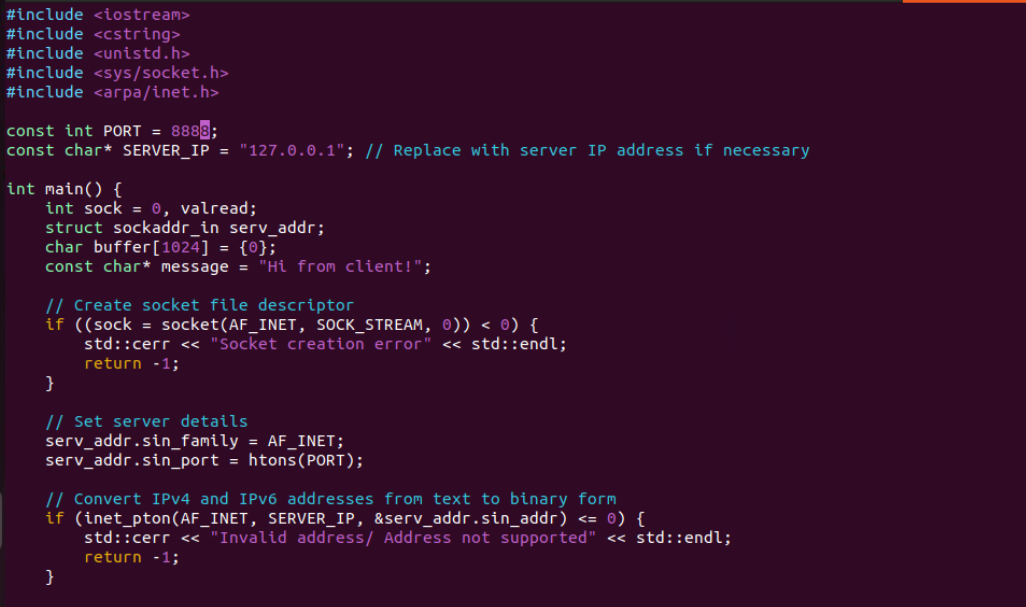
Simple TCP Server: Write a basic C++ program that acts as a server that listens for incoming TCP connections on a specific port. Upon receiving a connection, the server should send a predefined message (e.g., "Hello, client!") to the client and then close the connection. Utilize system calls like socket, bind, listen, accept, send, and recv. (Note: Network programming involves additional libraries/headers. Refer to system documentation).

Server code:





Client code:





Output:

