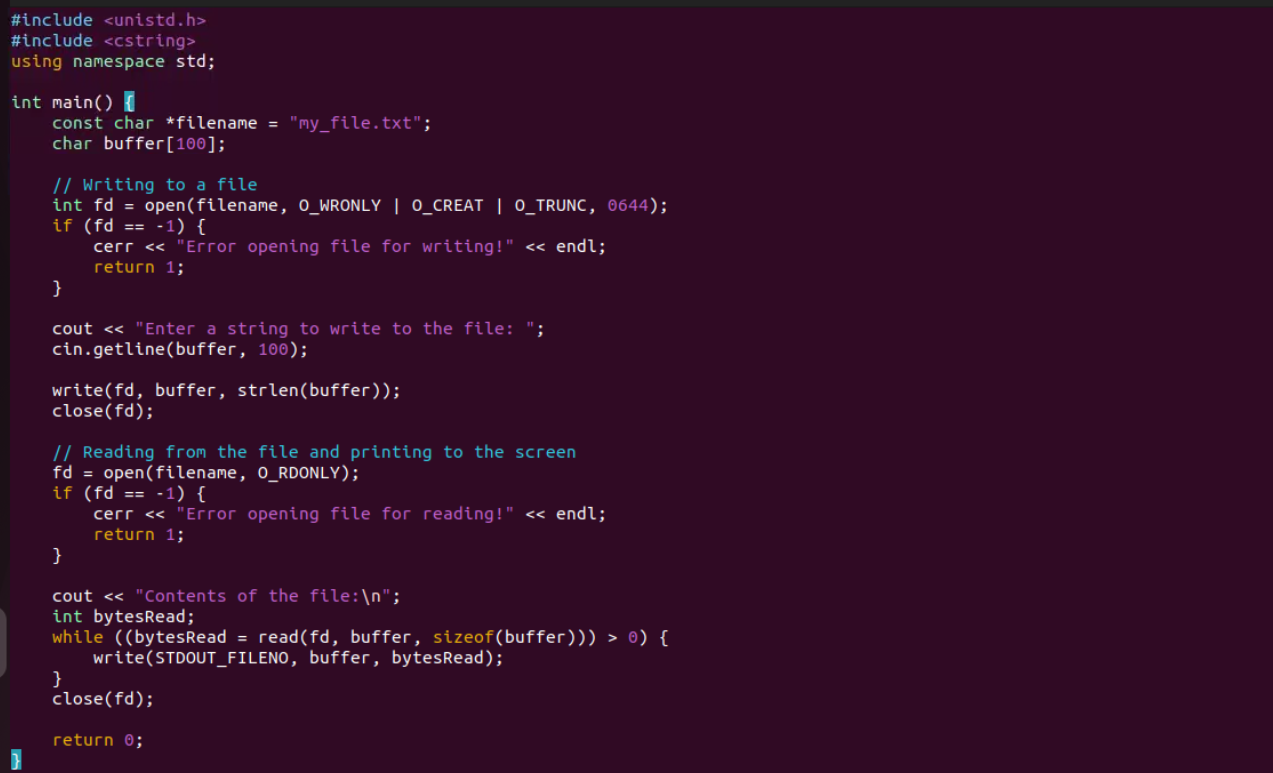
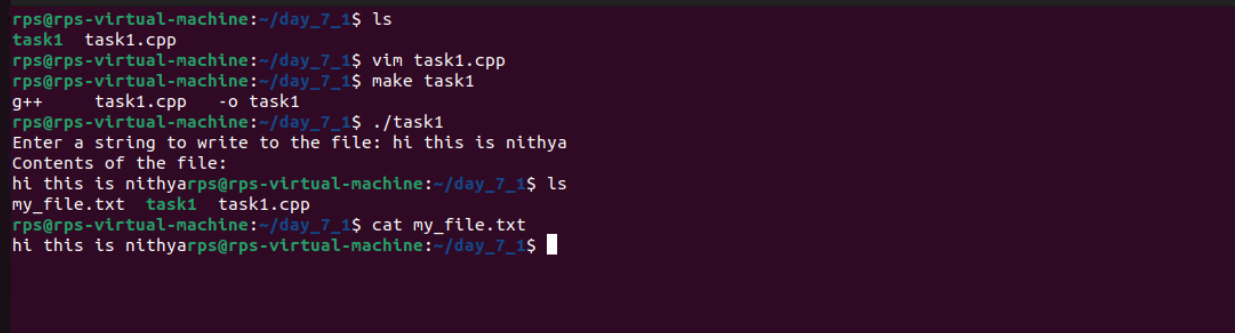
**01-08-24**

Task 1: write to a file

Code:



Output:



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

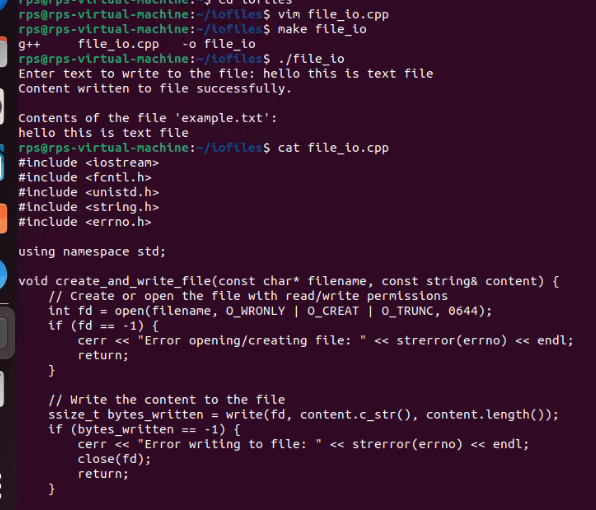
Specific Requirements:

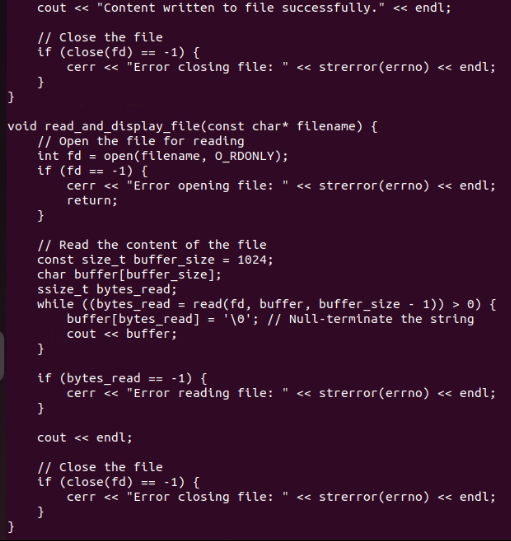
Create a new file if it doesn't exist.

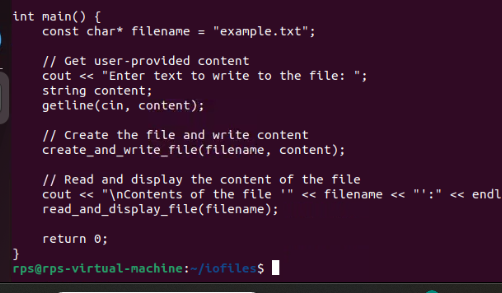
Write user-provided text content to the file.

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

Implement robust error handling for file operations.







**Short note on : man open, man close, man write, man read.**

1. **man open**

The open system call is used to open a file for reading or writing. It can also be used to create a new file if it doesn't exist.

**Usage:**

#include <fcntl.h>

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

**Parameters:**

* pathname: The name of the file to open.
* flags: Determines the file access mode (e.g., O\_RDONLY, O\_WRONLY, O\_RDWR) and file creation options (e.g., O\_CREAT, O\_TRUNC).
* mode: Specifies the permissions to use in case a new file is created (e.g., S\_IRUSR | S\_IWUSR).

**Returns:**

* On success, returns a file descriptor (a non-negative integer).
* On failure, returns -1 and sets errno to indicate the error.

**Example:**

int fd = open("file.txt", O\_WRONLY | O\_CREAT, S\_IRUSR | S\_IWUSR);

if (fd == -1) {

perror("open");

}

1. **man close**

The close system call is used to close an open file descriptor.

**Usage:**

#include <unistd.h>

int close(int fd);

**Parameters:**

* fd: The file descriptor to close.

**Returns:**

* On success, returns 0.
* On failure, returns -1 and sets errno to indicate the error.

**Example:**

if (close(fd) == -1) {

perror("close");

}

1. **man write**

The write system call is used to write data to an open file descriptor.

**Usage:**

#include <unistd.h>

ssize\_t write(int fd, const void \*buf, size\_t count);

**Parameters:**

* fd: The file descriptor to write to.
* buf: A pointer to the buffer containing the data to write.
* count: The number of bytes to write from the buffer.

**Returns:**

* On success, returns the number of bytes written.
* On failure, returns -1 and sets errno to indicate the error.

**Example:**

const char \*text = "Hello, World!";

ssize\_t bytes\_written = write(fd, text, strlen(text));

if (bytes\_written == -1) {

perror("write");

}

1. **man read**

The read system call is used to read data from an open file descriptor.

**Usage:**

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

**Parameters:**

* fd: The file descriptor to read from.
* buf: A pointer to the buffer where the read data will be stored.
* count: The number of bytes to read into the buffer.

**Returns:**

* On success, returns the number of bytes read (which may be less than count).
* On failure, returns -1 and sets errno to indicate the error.
* A return value of 0 indicates end-of-file (EOF).

**Example:**

char buffer[1024];

ssize\_t bytes\_read = read(fd, buffer, sizeof(buffer) - 1);

if (bytes\_read == -1) {

perror("read");

} else {

buffer[bytes\_read] = '\0'; // Null-terminate the buffer

printf("Read: %s\n", buffer);

}

These system calls form the foundation for file I/O operations in POSIX-compliant operating systems. They provide low-level control over file descriptors, allowing for efficient and flexible file manipulation.

Task :

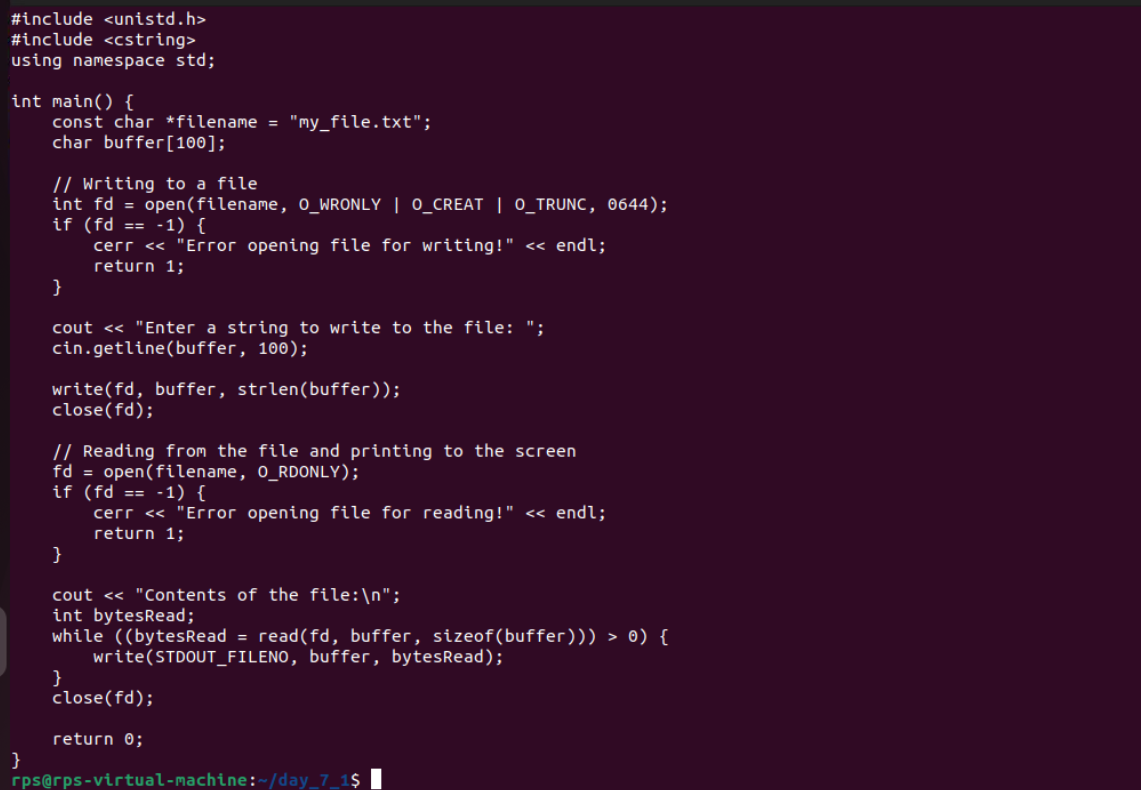
Gemini was just updated.  See update

Conversation with Gemini

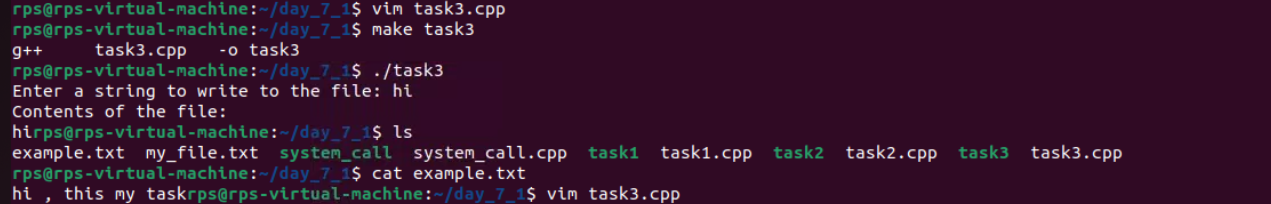
Write a c code use open close read and write to write onto file and print on screen using same

Show drafts

Code:

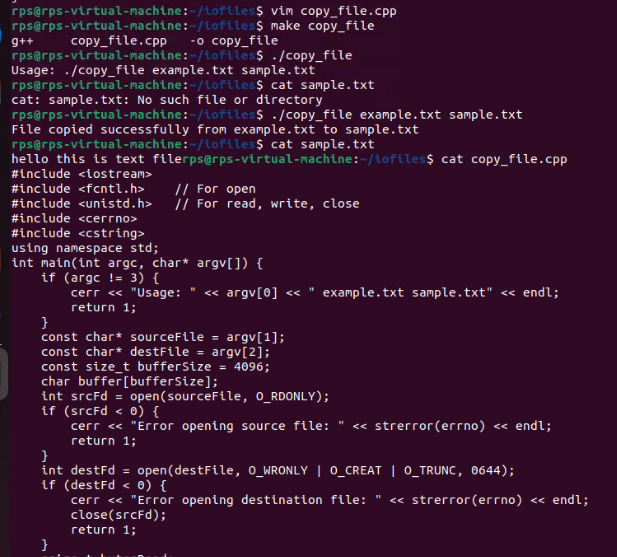


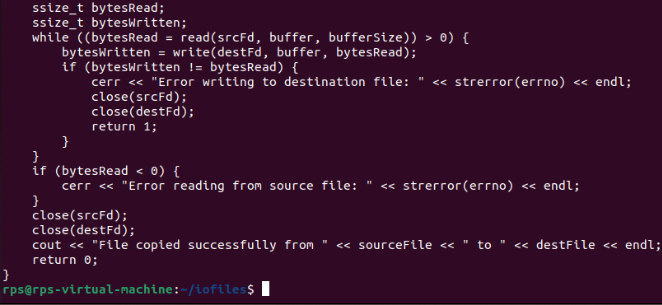
Output:



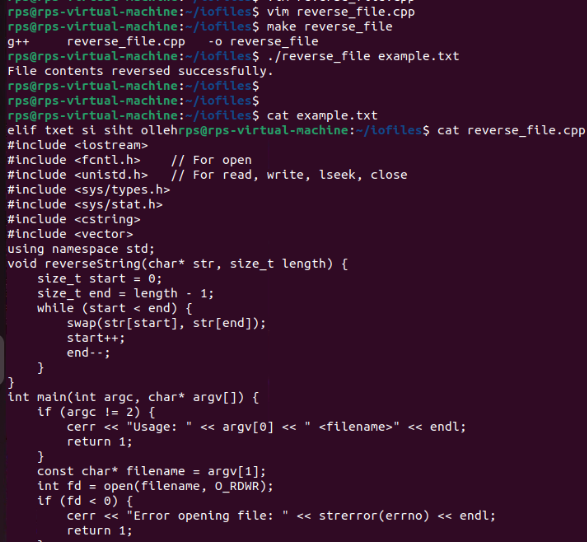
File I/O and Manipulation:

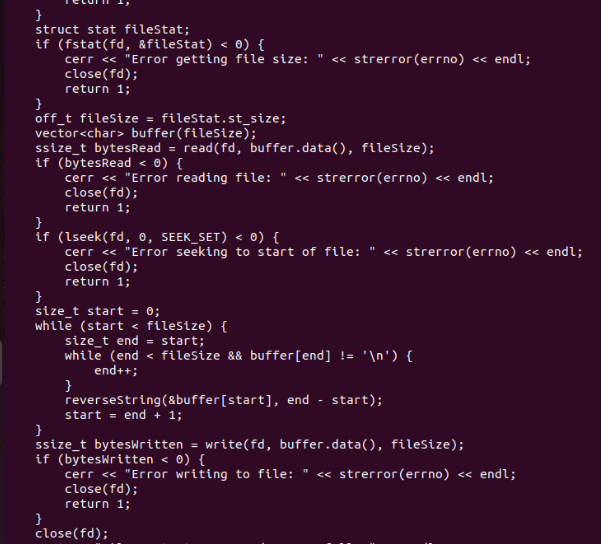
1. Copy a File: Write a C++ program that takes two file paths as command-line arguments. Use open, read, write, and close system calls to copy the contents of the source file to the destination file. Handle potential errors (e.g., file not found, permission denied).

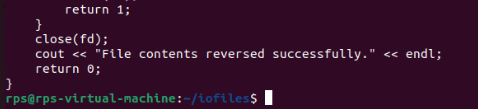




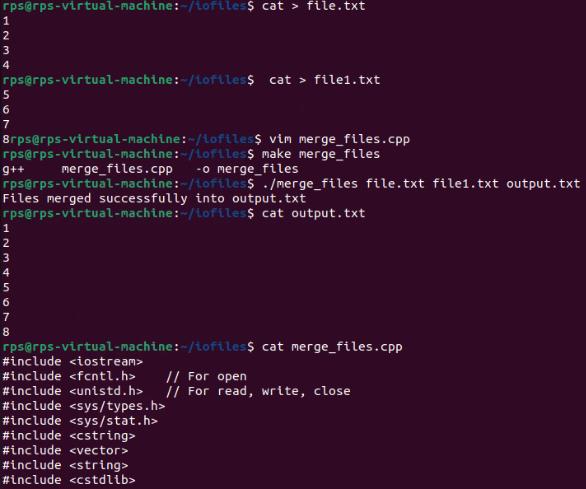
2. Reverse a File: Write a C++ program that reads the contents of a file line by line, reverses each line in-place, and then writes the reversed lines back to the same file. Use system calls like open, read, write, lseek, and close to achieve this.

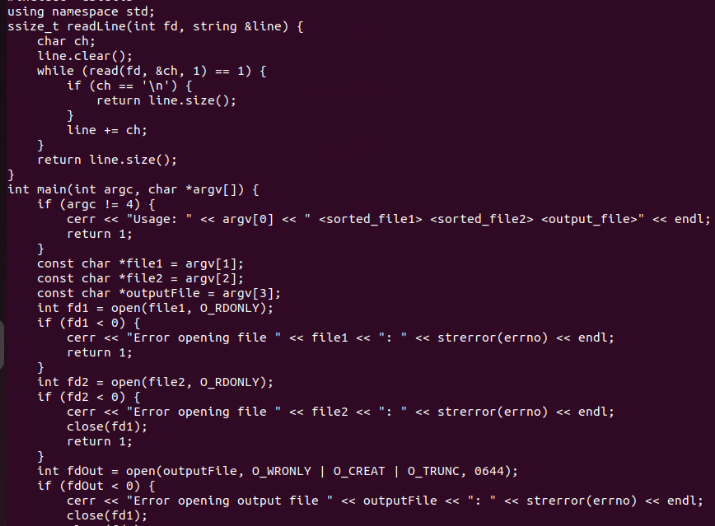






3. Merge Two Sorted Files: Write a C++ program that takes two sorted text files as input and creates a new file containing the merged and sorted contents. Use appropriate system calls for file handling and consider memory efficiency when handling large files.

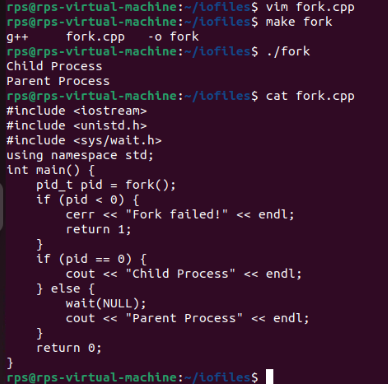




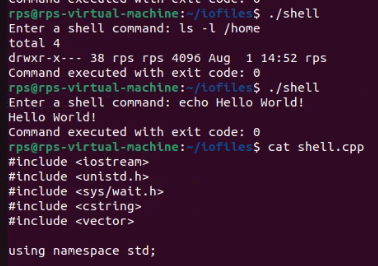


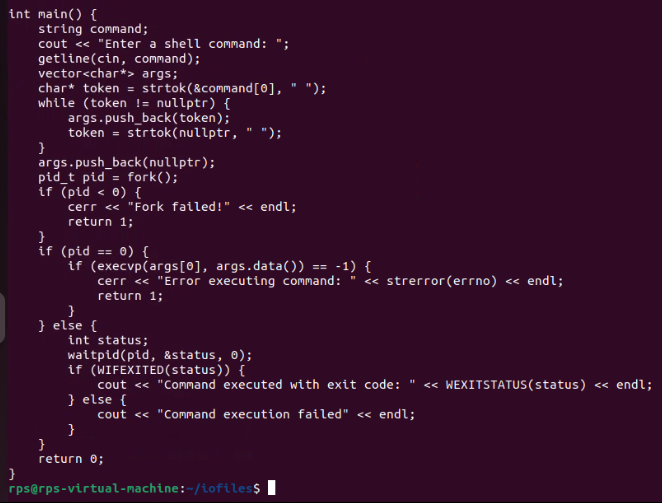
Process Control and Inter-Process Communication:

4. 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.

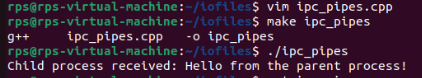


5. Execute a Shell Command: Write a C++ program that takes a shell command as a string argument and uses exec system calls (e.g., execlp or execv) to execute that command. Handle errors if the command execution fails.





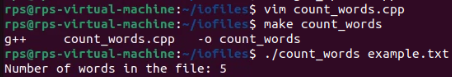
6. Inter-Process Communication with Pipes: Write a C++ program that demonstrates inter-process communication using pipes. One process should write data to a pipe, and another process should read from the pipe and print the received data. Leverage pipe and fork system calls.

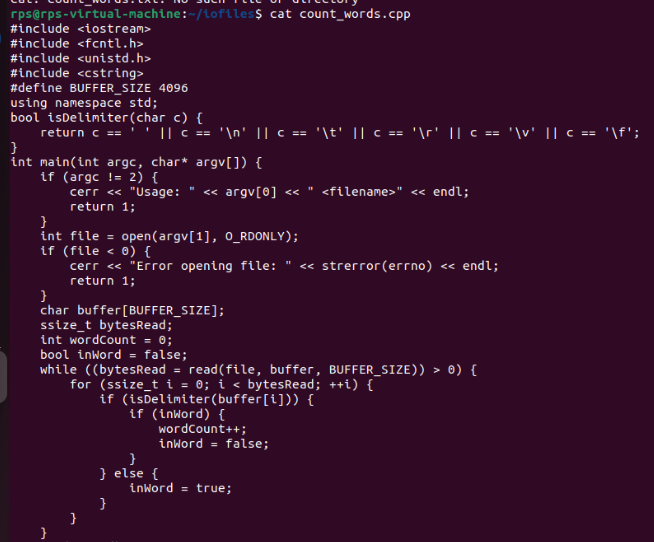


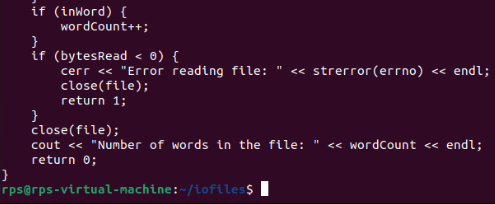


Text Processing and System Information:

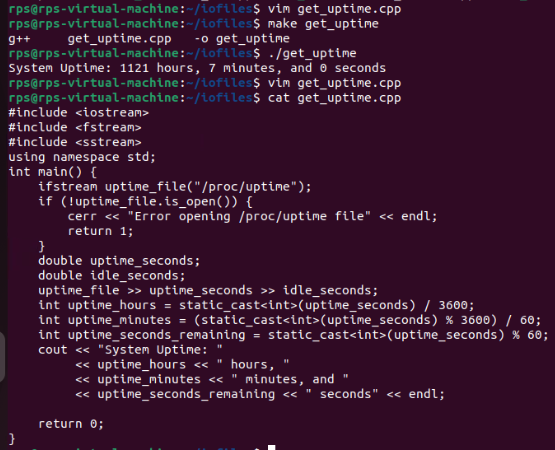
7. Count Words in a File: Write a C++ program that reads a text file and counts the number of words in it. Use open, read, and close system calls to access the file. Be mindful of delimiters and whitespace characters when counting words.





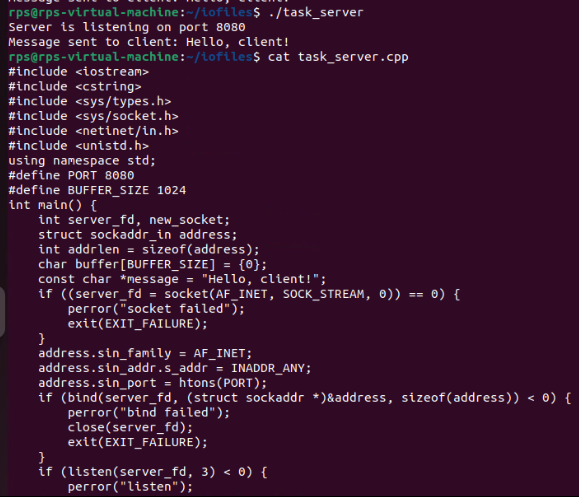


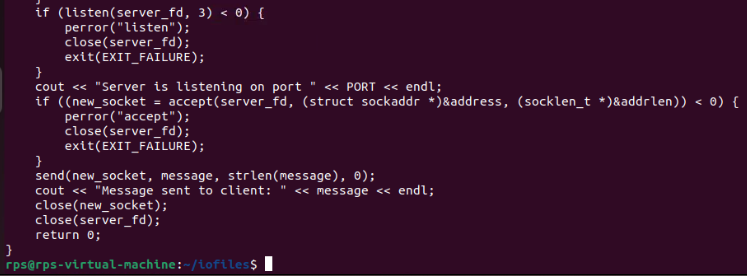
8. Get System Uptime: Write a C++ program that retrieves the system's uptime (time since it was last booted) using appropriate system calls (e.g., getuptime on Linux). Display the uptime information in a user-friendly format.

****

Network Programming (Bonus):

9. 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)





10. Simple TCP Client: As a companion to question 9, write a C++ program that acts as a client that connects to the server created in question 9. The client should send a message (e.g., "Hi from client!") to the server, receive the server's response, and then close the connection. (Note: Network programming details apply here as well)

