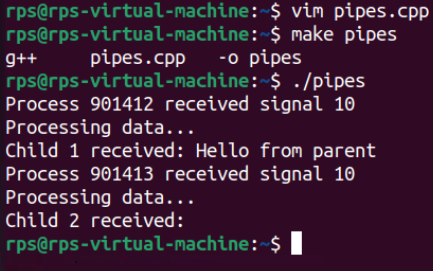
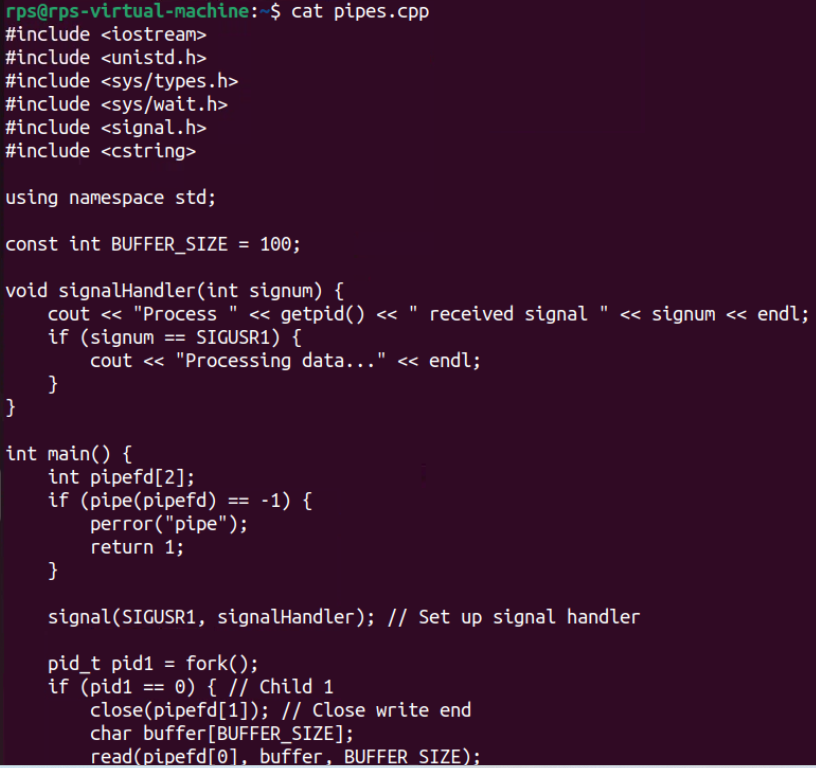
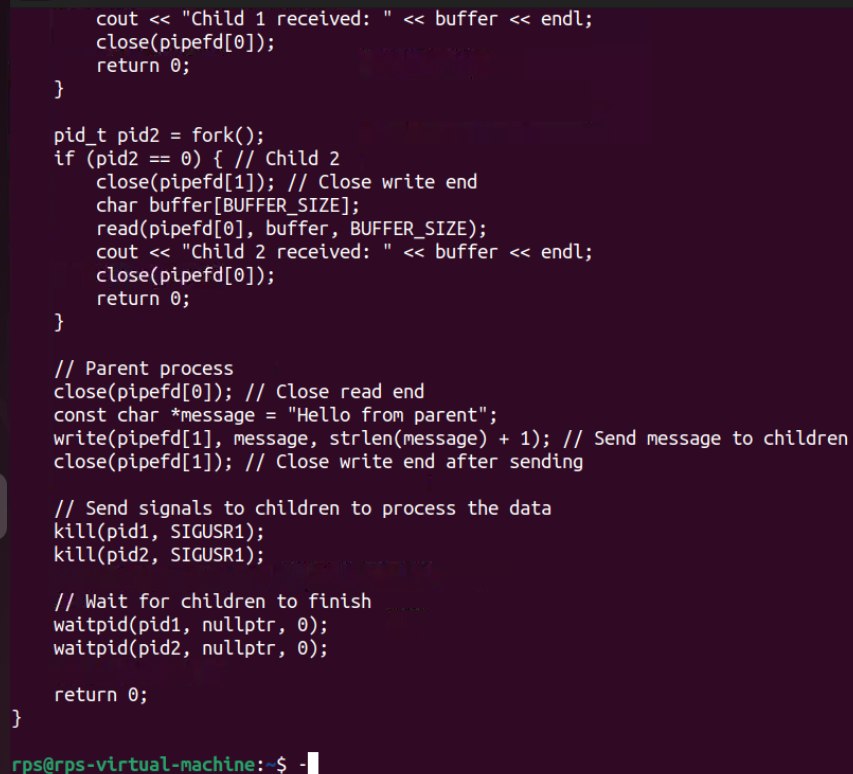
**Problem Statement: Signal Handling and Inter-Process Communication using Pipes in C++ Design and implement a robust system in C++ that effectively utilizes signals to control the behavior of multiple processes and employs pipes for inter-process communication, enabling coordinated data exchange and process synchronization**.







**l - execl(), execlp(), execle()**

The const char \*arg and subsequent ellipses can be thought of as arg0, arg1, ..., argn. Together they

describe a list of one or more pointers to null-terminated strings that represent the argument list

available to the executed program. The first argument, by convention, should point to the filename

associated with the file being executed. The list of arguments must be terminated by a null pointer,

and, since these are variadic functions, this pointer must be cast (char \*) NULL.

By contrast with the 'l' functions, the 'v' functions (below) specify the command-line arguments of

the executed program as a vector.

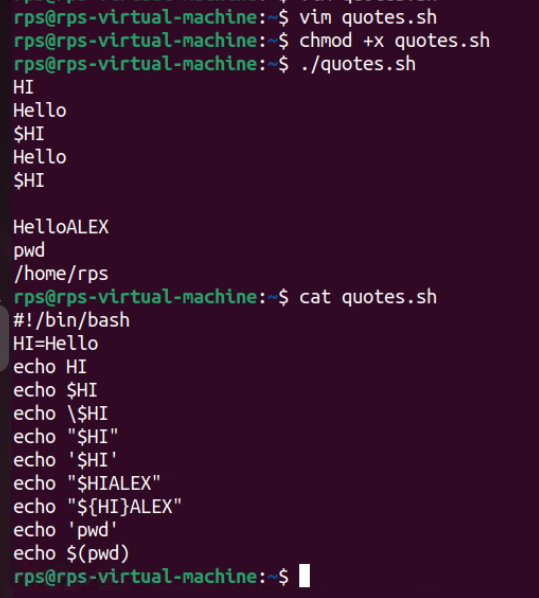
**v - execv(), execvp(), execvpe()**

The char \*const argv[] argument is an array of pointers to null-terminated strings that represent the

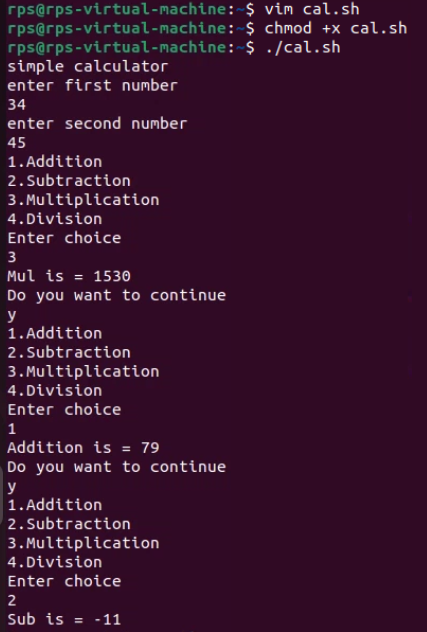
argument list available to the new program. The first argument, by convention, should point to the

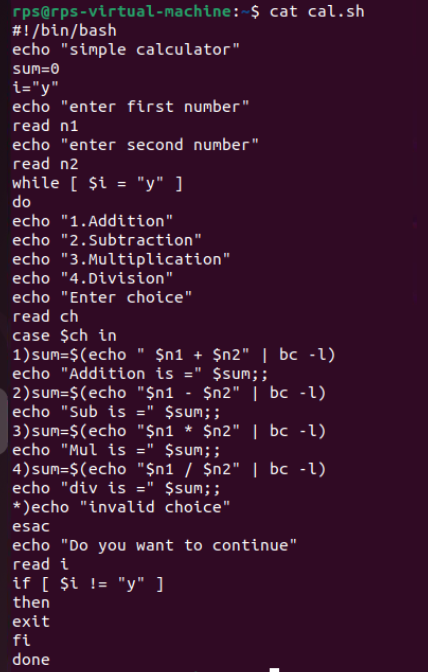
filename associated with the file being executed. The array of pointers must be terminated by a null pointer.

**Shell programs:**



**Calculator:**





**Change File Permissions**

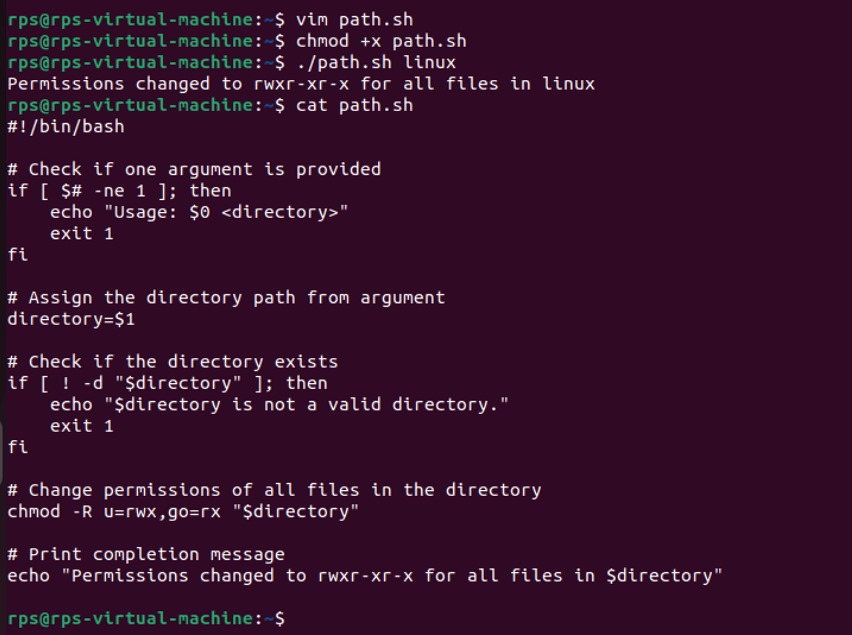
**Description: Write a shell script that takes a directory path as an argument and changes the permissions of all files within that directory to read, write, and execute for the owner, and read and execute for the group and others.**

**Instructions:**

**The script should accept one argument, the directory path.**

**Change permissions of all files in the specified directory to rwxr-xr-x.**

**Print a message indicating the completion of the permission change.**



**Problem 2: Count Files and Directories**

**Description: Write a shell script that counts the number of files and directories in a given directory.**

**Instructions:**

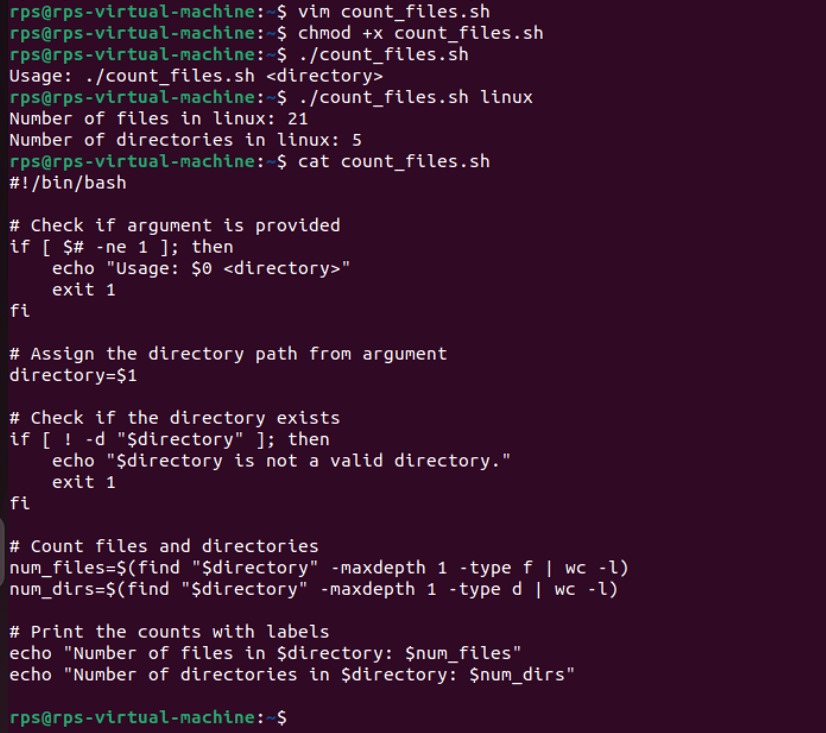
**The script should accept one argument, the directory path.**

**Count the number of files and directories separately.**

**Print the counts with appropriate labels.**

**Sample Input:**

**./count\_files\_dirs.sh /path/to/directory**



**Problem 3: Find and Replace Text in Files**

**Description: Write a shell script to search for a specific text string in all files within a directory and replace it with another string.**

**Instructions:**

**The script should accept three arguments: directory path, search string, and replacement string.**

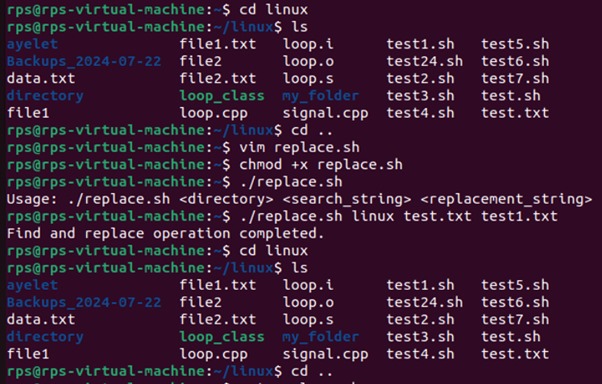
**Search for the specified string in all files within the directory.**

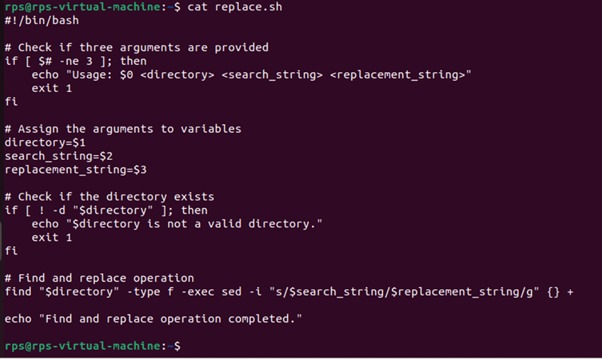
**Replace the string with the given replacement string in all occurrences.**

**Print a message indicating the completion of the find and replace operation.**

**Sample Input:**

**./find\_replace.sh /path/to/directory "old\_text" "new\_text"**





**Problem 4: Disk Usage Report**

**Description: Write a shell script that generates a report of disk usage for a specified directory.**

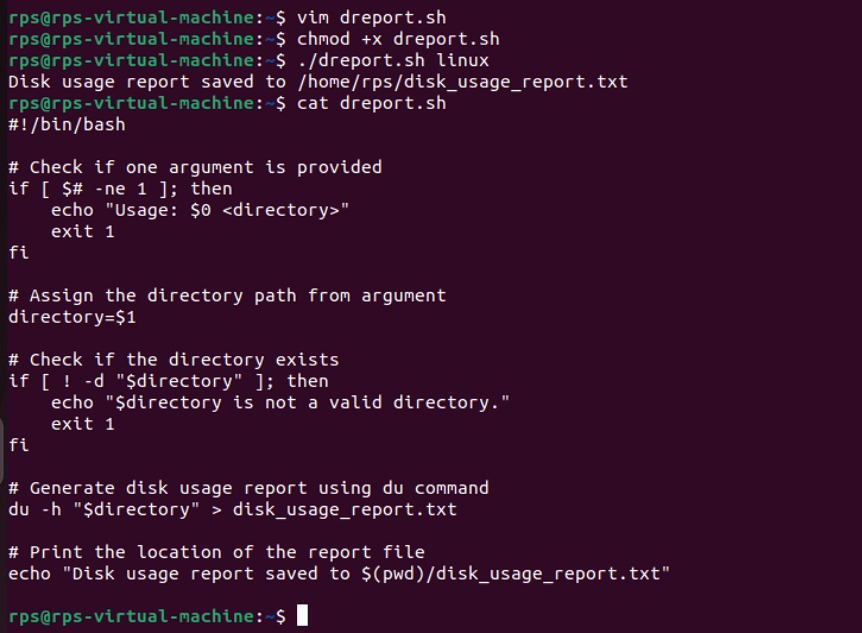
**Instructions:**

**The script should accept one argument, the directory path.**

**Use the du command to generate a disk usage report for the directory.**

**Save the report to a file named disk\_usage\_report.txt in the current directory.**

**Print a message indicating where the report is saved.**



**Problem Statement: File Management Script with Functions and Arguments**

**Objective**

**Create a shell script that manages files in a specified directory. The script should include functions to perform the following tasks:**

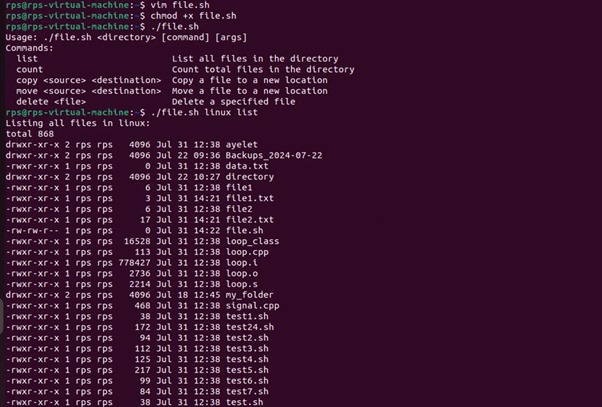
**List all files in the directory.**

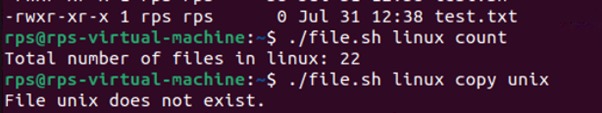
**Display the total number of files.**

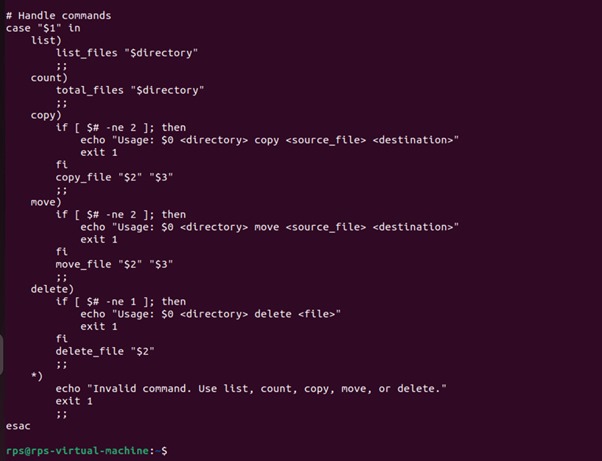
**Copy a specified file to a new location.**

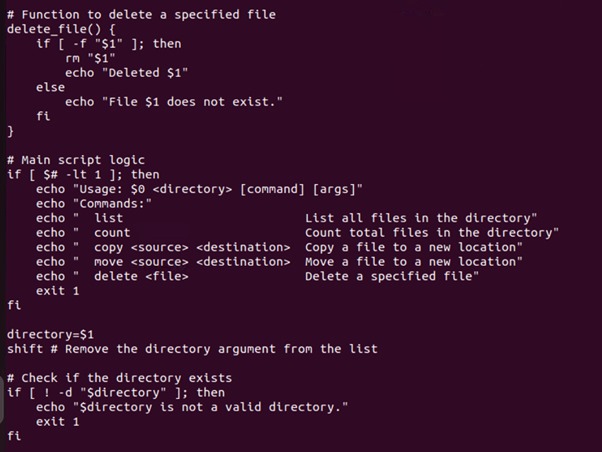
**Move a specified file to a new location.**

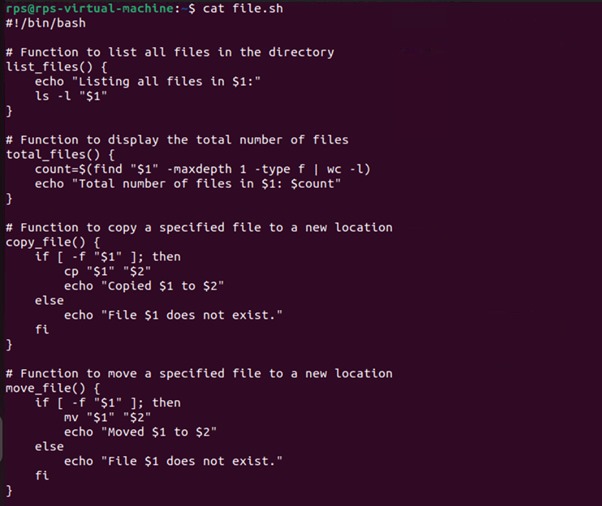
**Delete a specified file.**



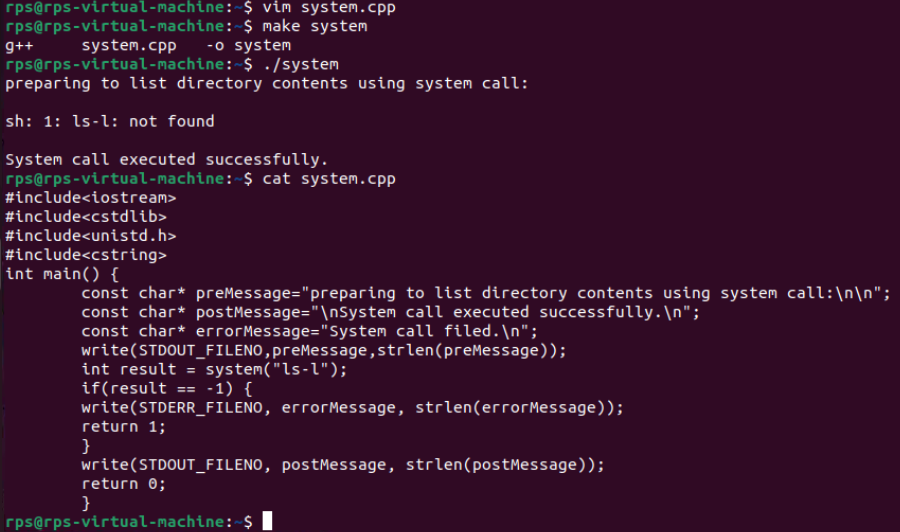




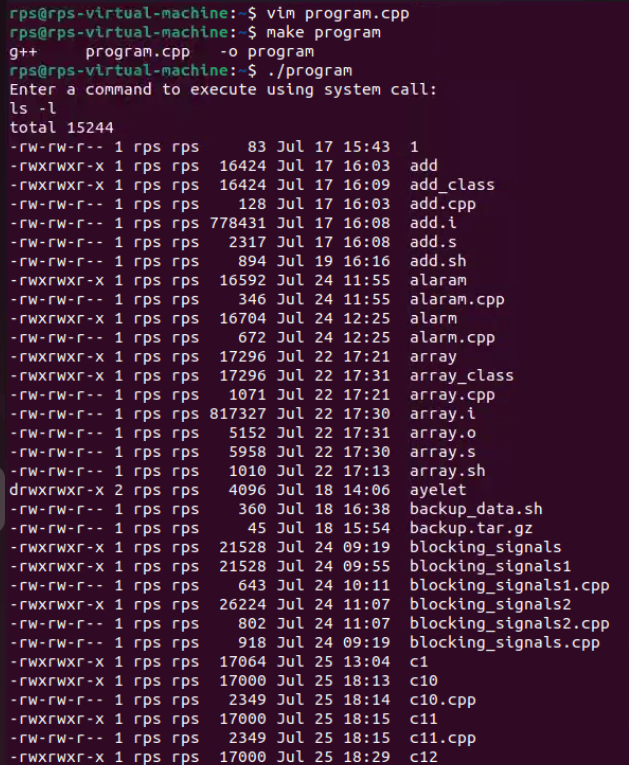


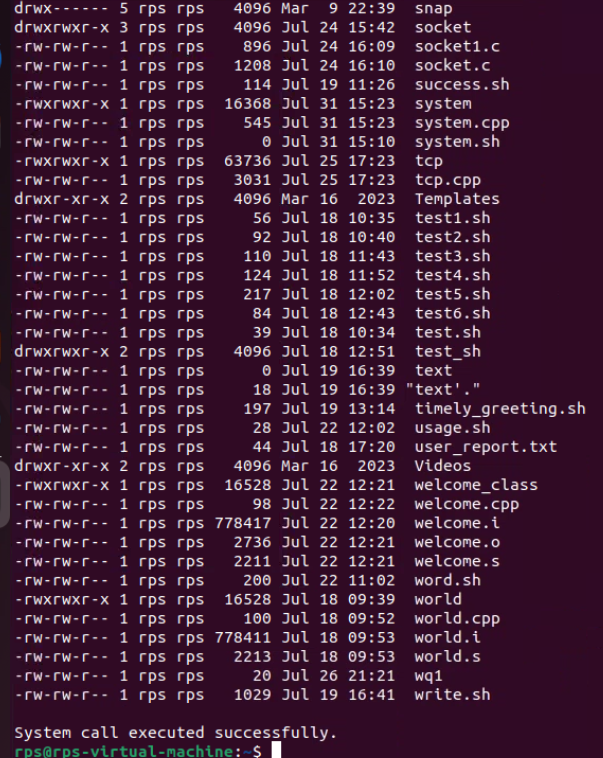


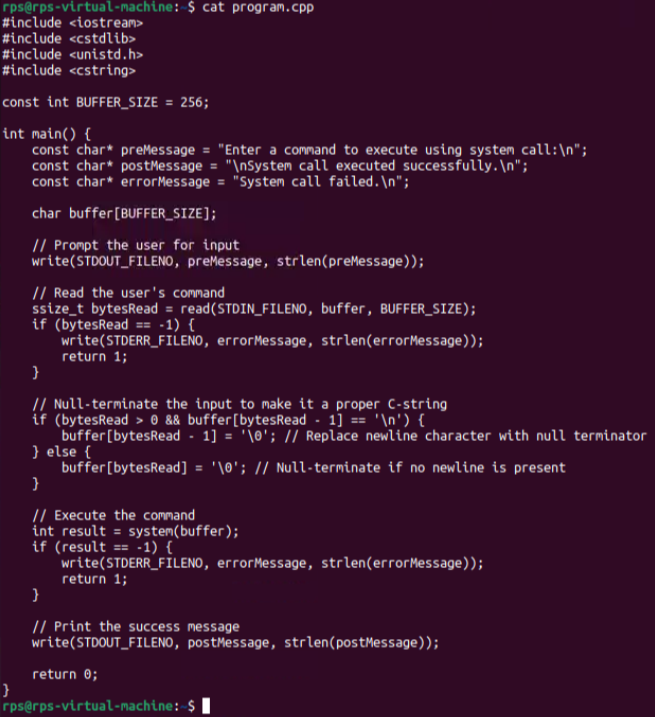
**System call:**



**read from user and write on screen using read and write apis in cpp**







**Problem Statement: File Operations using System Calls in C++**

**Description:**

**Write a C++ program that performs various file operations using Linux system calls. The program should create a file, write to it, read from it, and then delete the file. The program should handle errors appropriately and ensure proper resource management (e.g., closing file descriptors).**

**Instructions:**

**Create a File:**

**Use the open system call to create a new file named "example.txt" with read and write permissions.**

**If the file already exists, truncate its contents.**

**Write to the File:**

**Write the string "Hello, World!" to the file using the write system call.**

**Ensure that all bytes are written to the file.**

**Read from the File:**

**Use the lseek system call to reset the file pointer to the beginning of the file.**

**Read the contents of the file using the read system call and store it in a buffer.**

**Print the contents of the buffer to the standard output.**

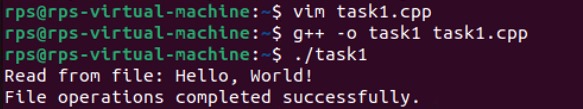
**Delete the File:**

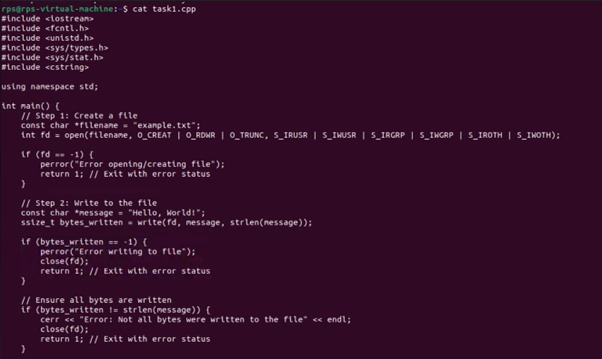
**Close the file descriptor using the close system call.**

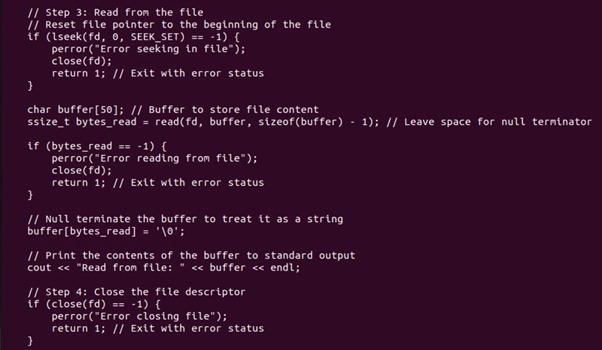
**Use the unlink system call to delete the file "example.txt".**

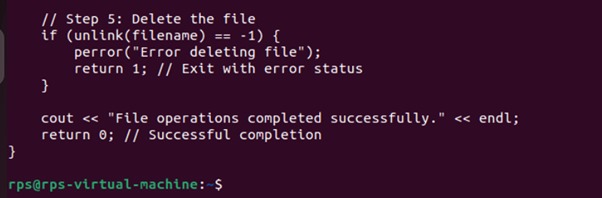
**Error Handling:**

**Ensure proper error handling for each system call. If a system call fails, print an error message and exit the program with a non-zero status.**









**Question:**

**Explain the role of virtual memory in Linux memory management. How does the kernel use system calls like brk, mmap, and munmap to manage virtual memory for processes? Discuss the implications of overcommitting memory and the mechanisms Linux employs to handle memory pressure.**

**Potential Areas for Further Exploration:**

**Deep dive into specific system calls: Explore the inner workings of brk, mmap, and munmap in detail, including their parameters, return values, and common use cases.**

**Memory allocation algorithms: Discuss different memory allocation strategies used by the kernel, such as the buddy system and slab allocator.**

**Performance implications: Analyze the performance impact of different memory management techniques under various workloads.**

**Memory management in specific scenarios: Explore memory management challenges and solutions in specific use cases like containerization or real-time systems.**

**Role of Virtual Memory in Linux Memory Management**

Virtual memory is an essential abstraction that allows the Linux operating system to provide each process with its own isolated address space. This abstraction serves several critical purposes:

Isolation: Each process operates in its own virtual address space, preventing one process from accessing the memory of another. This isolation enhances security and stability by preventing accidental or malicious interference between processes.

Efficient Memory Utilization: Virtual memory allows processes to use more memory than is physically available by leveraging disk space as "swap." This enables the execution of larger applications and allows the system to run more processes concurrently.

Simplified Memory Management: Programmers can allocate memory without worrying about the underlying physical memory constraints. The operating system handles the mapping between virtual addresses and physical memory addresses.

System Calls for Managing Virtual MemoryLinux provides various system calls for managing virtual memory, with the most notable being brk, mmap, and munmap. Each of these serves a specific role in the memory management process.

1. **brk and sbrk**

Purpose: The brk and sbrk system calls are used to manage memory allocation in the heap. They allow dynamic memory allocation and deallocation for a process's data segment.

Parameters:

brk(void \*end\_data\_segment): Sets the end of the data segment (heap) to the specified address.

sbrk(int increment): Increments the program's data segment by a specified number of bytes (positive for allocation, negative for deallocation).

Return Value: Both return the previous end of the data segment on success, or (void \*) -1 on failure.

Common Use Case: These calls are often used by the C standard library functions like malloc and free for dynamic memory management.

1. **mmap**

Purpose: The mmap system call maps files or devices into memory and allocates memory regions. It is more flexible than brk since it allows mapping of non-contiguous physical memory.

Parameters:

mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset):

addr: Suggested starting address for the mapping (can be NULL for automatic selection).

length: Length of the mapping.

prot: Protection flags (e.g., PROT\_READ, PROT\_WRITE).

flags: Mapping options (e.g., MAP\_PRIVATE, MAP\_SHARED).

fd: File descriptor for the file being mapped (if applicable).

offset: Offset within the file from which the mapping starts.

Return Value: Returns the starting address of the mapped area on success, or (void \*) -1 on failure.Common Use Cases: mmap is used for memory-mapped file I/O, shared memory, and for allocating large chunks of memory that might exceed the limits of brk.3. munmap

Purpose: The munmap system call unmaps a previously mapped memory region. Parameters:

munmap(void \*addr, size\_t length): Unmaps the specified range of memory.

Return Value: Returns 0 on success or -1 on failure.

Common Use Case: Used to free up memory that was allocated with mmap.

**Implications of Overcommitting Memory**

Linux employs a strategy for memory overcommitment, which allows it to allocate more memory than is physically available based on the assumption that not all allocated memory will be used at once.

Overcommit Behavior:

Default (0): The kernel allows overcommitment, which means it can allocate memory based on heuristics that predict usage.

Never (2): Memory allocations are only allowed if enough physical memory is available.

Always (1): Memory allocation requests are always granted, regardless of available memory.

Consequences: Overcommitting can lead to scenarios where processes consume more memory than the system can physically support, potentially leading to system instability and invoking the Out of Memory (OOM) killer, which terminates processes to reclaim memory.

**Mechanisms to Handle Memory Pressure**

When the system experiences memory pressure, the Linux kernel employs several strategies to manage memory effectively:

Paging: The kernel can swap pages of memory to disk when physical memory is low. This allows the system to free up RAM by moving inactive pages to a swap space.

Out of Memory (OOM) Killer: When the system is critically low on memory, the OOM killer selects and terminates processes based on their memory usage patterns to recover memory resources.

Swappiness: This kernel parameter controls the balance between using swap space and retaining active processes in RAM. A higher swappiness value encourages the kernel to swap out more pages, while a lower value keeps more pages in RAM.

**Potential Areas for Further Exploration**

1. **Deep Dive into System Calls:**

brk and sbrk: Explore how these functions interface with the memory allocation routines in libraries like glibc, and their performance implications during rapid allocation and deallocation.

mmap: Investigate how mmap supports file-backed memory and anonymous memory, and analyze its impact on system performance, particularly in I/O operations.

1. **Memory Allocation Algorithms:**

Buddy System: Discuss how the buddy memory allocation system efficiently manages free memory blocks and its advantages and disadvantages in terms of fragmentation.

Slab Allocator: Explore the slab allocator’s design for kernel memory management, focusing on its efficiency for small objects and reducing fragmentation.

1. Performance Implications:

Analyze the performance impact of various memory management techniques under different workloads, considering factors such as latency, throughput, and resource contention.

1. **Memory Management in Specific Scenarios:**

Containerization: Investigate how Linux manages memory for containerized applications (e.g., Docker) and the challenges that arise from resource sharing.

Real-time Systems: Explore memory management requirements in real-time operating systems, where predictable performance is critical, and how Linux can be configured to support such use cases.