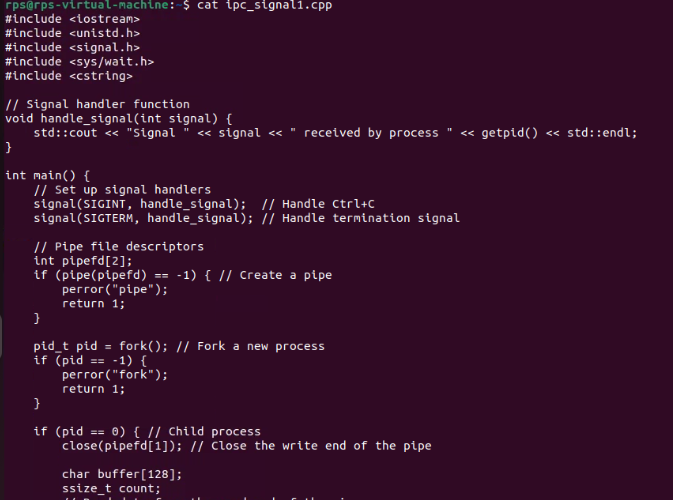
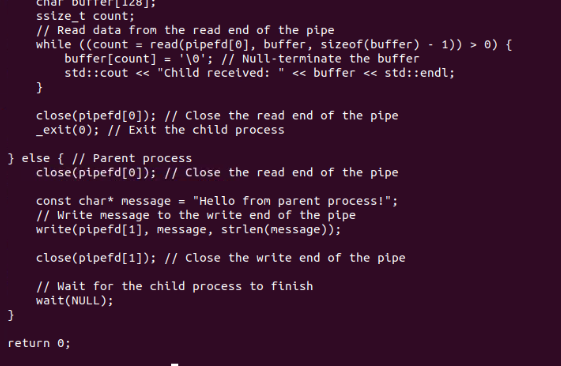
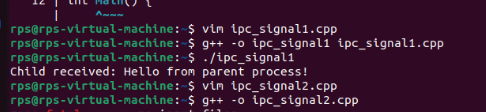
Problem Statement: Signal Handling and Inter-Process Communication using Pipes in C++

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.  
Code:

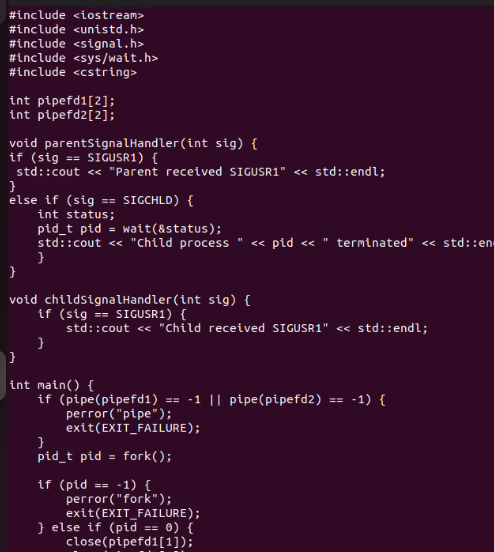


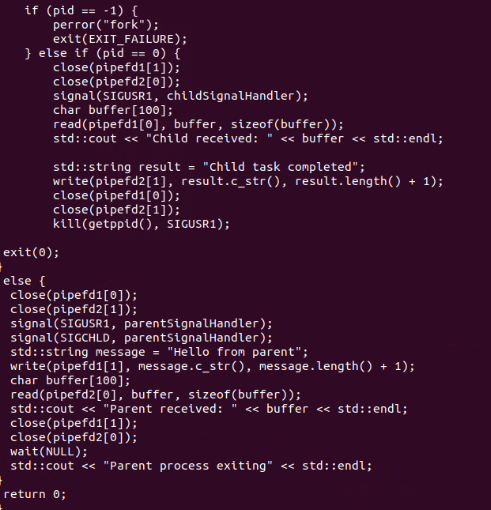


Output:

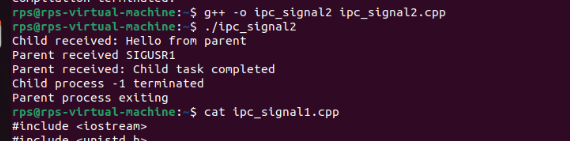


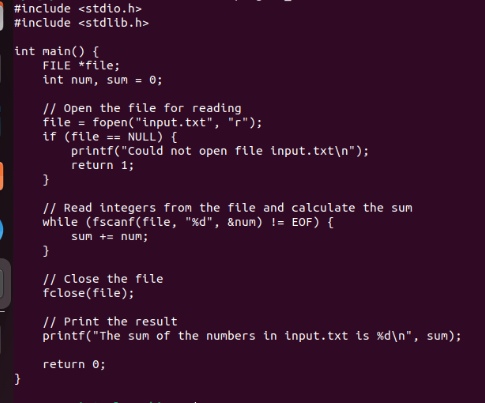
Part2:

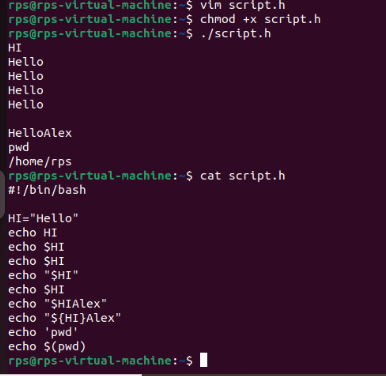




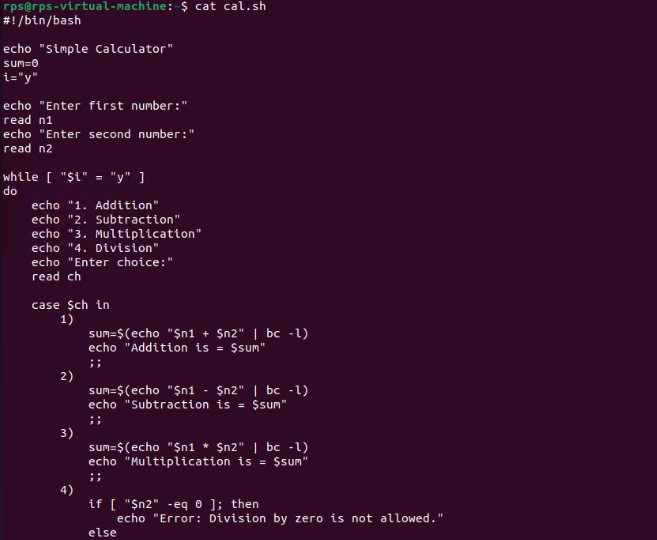
Output:

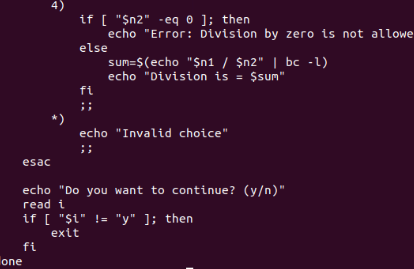




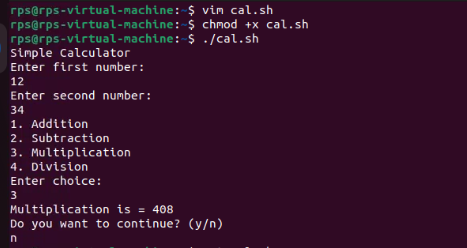


Calculator:





Output:



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.

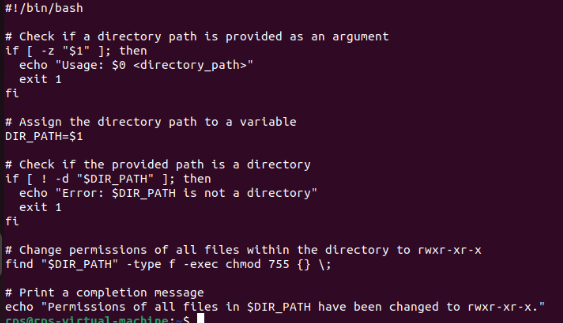
Instructions:

The script should accept one argument, the directory path.

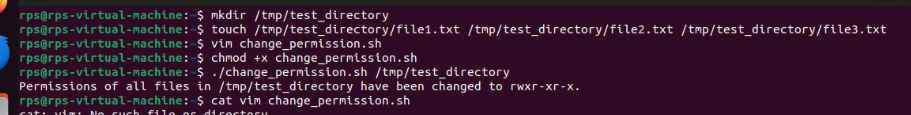
Change permissions of all files in the specified directory to rwxr-xr-x.

Print a message indicating the completion of the permission change.

Code:



Output:



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.

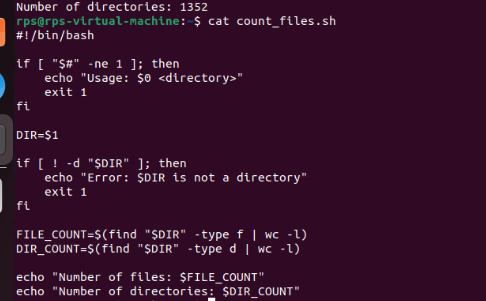
Count the number of files and directories separately.

Print the counts with appropriate labels.

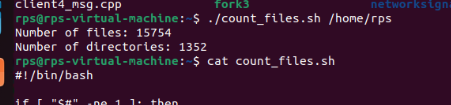
Sample Input:

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

Code



Output:



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.

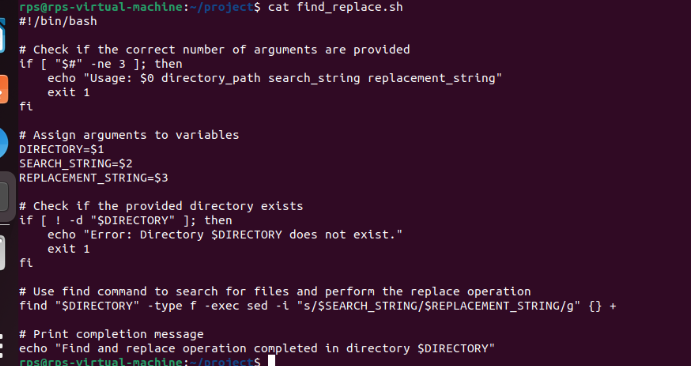
Search for the specified string in all files within the directory.

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

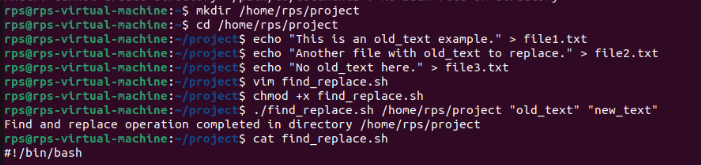
Print a message indicating the completion of the find and replace operation.

Sample Input:

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



Output:



Problem 4: Disk Usage Report

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

Instructions:

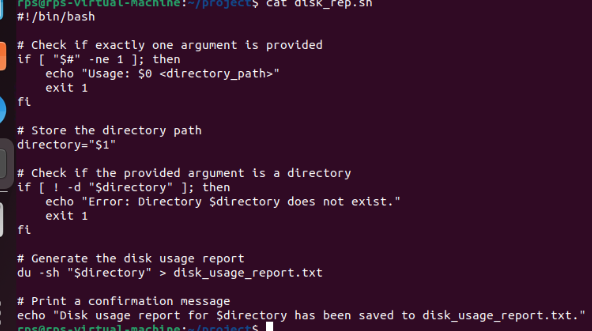
The script should accept one argument, the directory path.

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

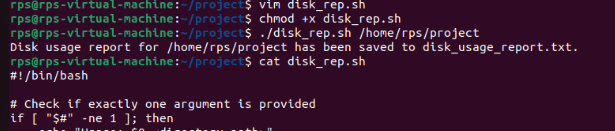
Save the report to a file named disk\_usage\_report.txt in the current directory.

Print a message indicating where the report is saved.

Code:



Output:



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.

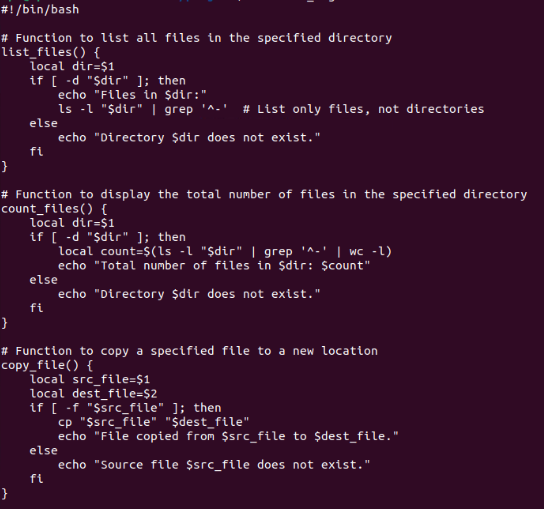
Display the total number of files.

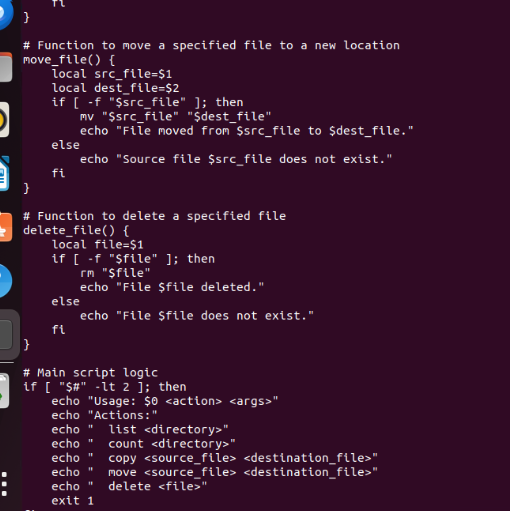
Copy a specified file to a new location.

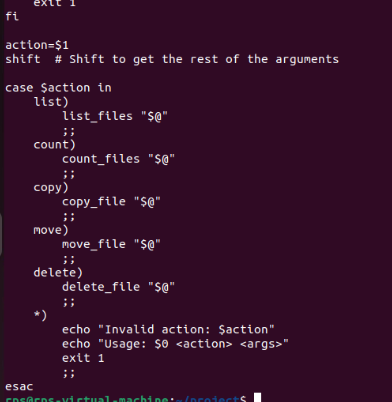
Move a specified file to a new location.

Delete a specified file.

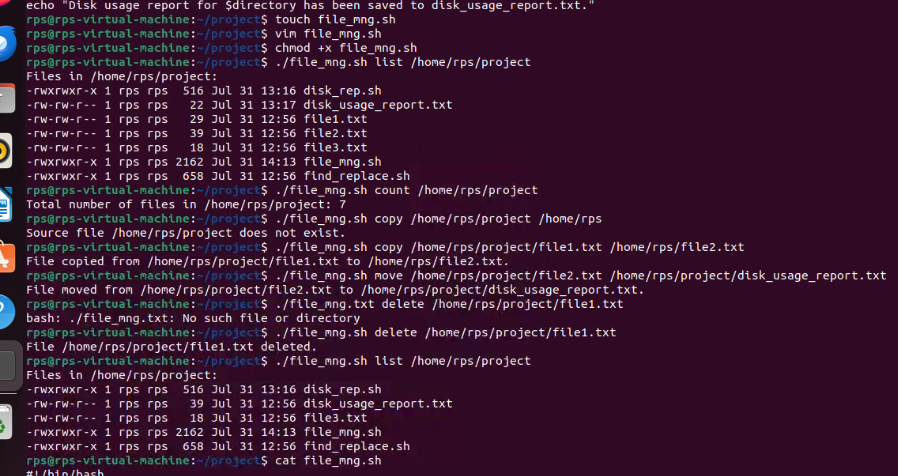
Code:



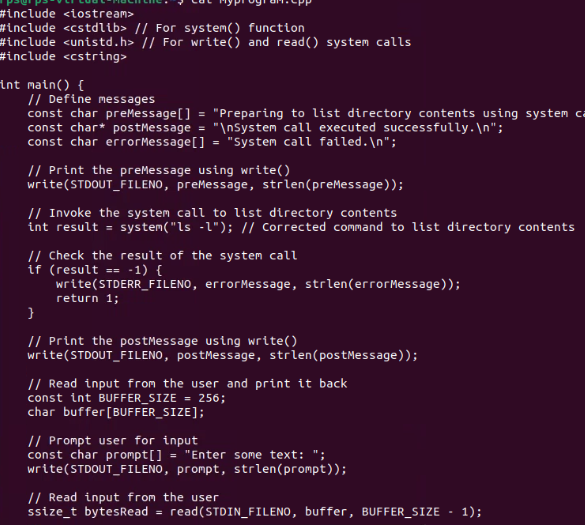


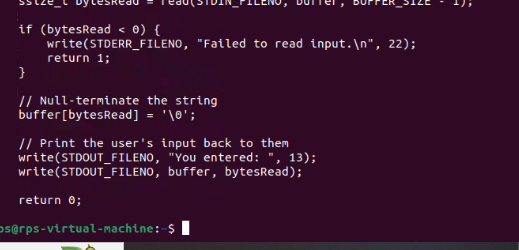


Output:

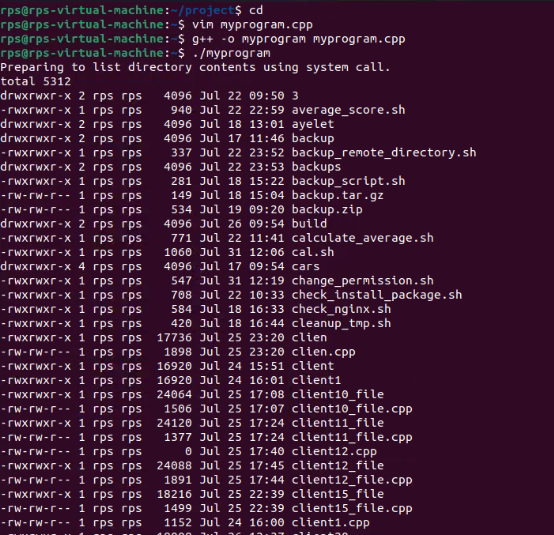


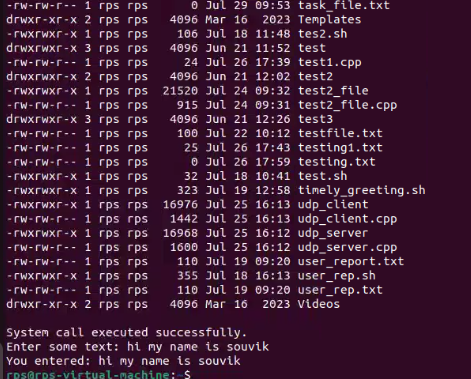
after writing the code please read from user and write on screen using read and write apis in cpp.





Output:





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:

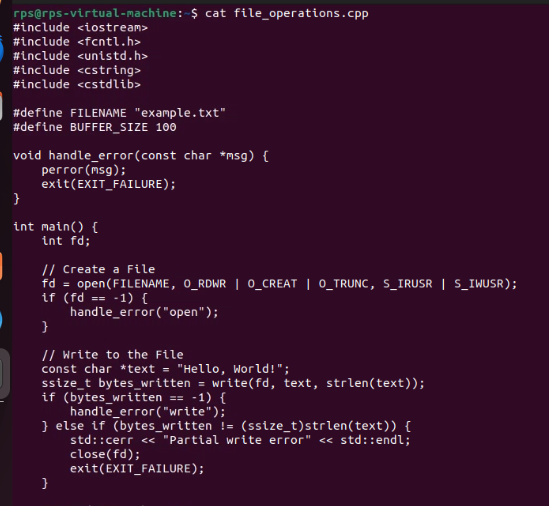
Close the file descriptor using the close system call.

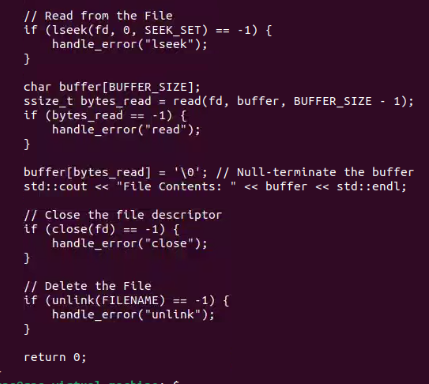
Use the unlink system call to delete the file "example.txt".

Error Handling:

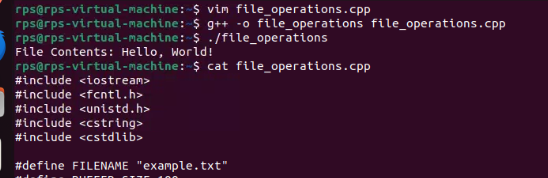
Ensure proper error handling for each system call. If a system call fails, print an error message and exit the program with a non-zero status.

Code:





Output:



Explain the role of virtual memory in Linux memory management?

Virtual memory in Linux abstracts physical memory, providing each process with its own virtual address space, ensuring isolation and security. It allows efficient memory usage through demand paging and swapping, supports overcommitment, and simplifies memory management. Additionally, it enables shared memory for efficient inter-process communication and enhances system security through mechanisms like Address Space Layout Randomization (ASLR).

How does the kernel use system calls like brk, mmap, and munmap to manage virtual memory for processes?

The Linux kernel uses system calls like brk, mmap, and munmap to manage virtual memory for processes. Here's a brief explanation of each:

1. Brk
2. **Purpose**: Adjusts the end of the data segment (heap) for a process.
3. **Usage**: The brk system call and its companion sbrk are used to allocate or deallocate memory by changing the program break (the end of the heap).
4. **Function**: When a process requests more heap memory, brk increases the program break; when it frees memory, brk decreases it.
5. Mmap
6. **Purpose**: Maps files or devices into memory.
7. **Usage**: The mmap system call is used to allocate memory by mapping files or anonymous memory (memory not associated with any file) into the process's address space.
8. **Function**: It provides more flexibility than brk, allowing for non-contiguous memory allocation, shared memory mapping, and fine-grained control over memory regions.
9. Munmap
10. **Purpose**: Unmaps memory regions.
11. **Usage**: The munmap system call is used to deallocate memory that was previously allocated with mmap.
12. **Function**: It removes memory mappings, releasing the associated virtual memory and resources, making the address space available for other uses

Discuss the implications of overcommitting memory and the mechanisms Linux employs to handle memory pressure.

Implications of Overcommitting Memory

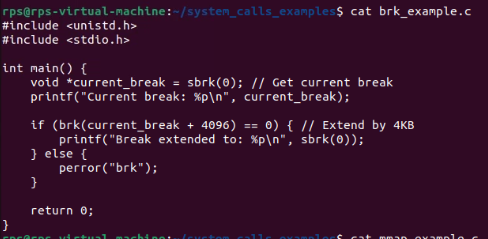
1. Overcommitting memory allows processes to allocate more memory than physically available, risking an Out-Of-Memory (OOM) situation.
2. If actual memory usage exceeds physical memory, it can lead to system instability and process termination.

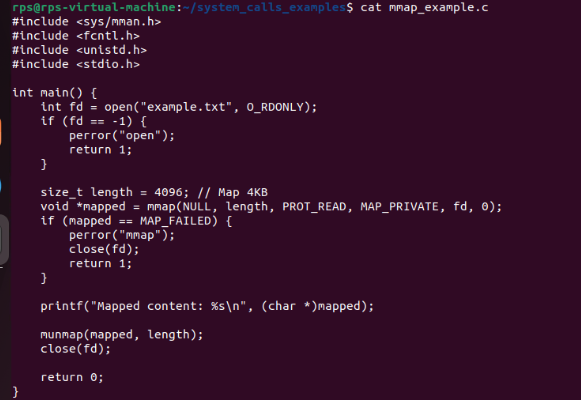
Mechanisms to Handle Memory Pressure

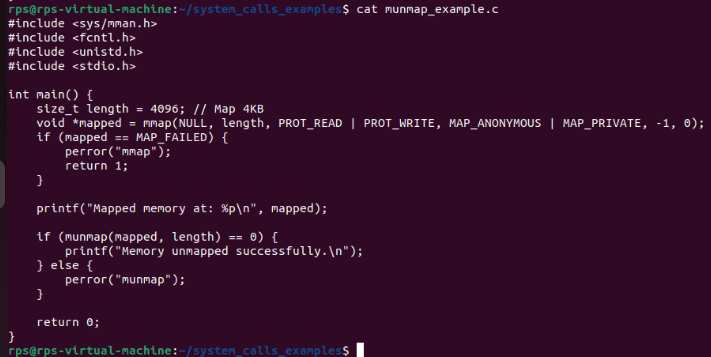
1. **OOM Killer**:
2. Terminates processes to free up memory when the system runs out of memory.
3. Targets processes based on factors like memory usage and priority.
4. **Swapping**:
5. Moves inactive pages from RAM to swap space (disk) to free up physical memory.
6. Helps manage short-term memory pressure but can degrade performance if overused.

Potential Areas for Further Exploration:

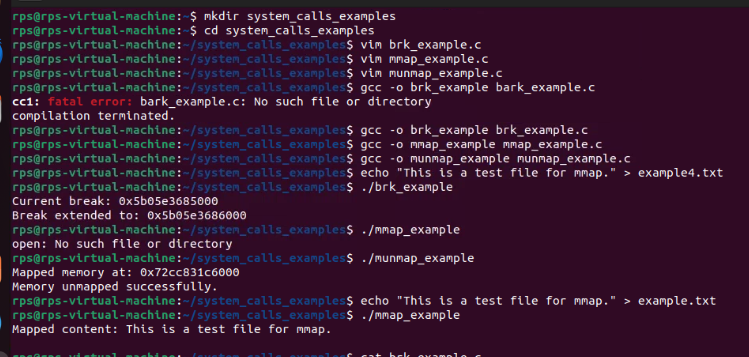
Deep dive into specific system calls: Explore the inner workings of brk, mmap, and munmap in detail, including their parameters, return values, and common use cases.  
code:







Output:



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

Buddy System

1. The memory is divided into blocks of size 2k2^k2k, where kkk is the order of the block. For instance, blocks of order 0 are 1 unit in size, blocks of order 1 are 2 units, and so forth.
2. When a memory request is made, the system searches for the smallest block that can accommodate the request. If a larger block is found, it is split into two smaller "buddy" blocks until a block of the appropriate size is obtained. Conversely, when a block is freed, the system checks if its buddy (the adjacent block) is also free. If so, the two buddies are coalesced into a larger block, and this process continues up the hierarchy.

Slab Allocator

1. The slab allocator maintains caches of pre-allocated memory blocks, called slabs, which are divided into equal-sized chunks. Each cache is tailored for a specific type of object (e.g., file descriptors, process control blocks).
2. When an object of a particular type is needed, the allocator first checks the corresponding cache. If there’s a free chunk, it’s allocated to the request. If no free chunks are available, a new slab is allocated and added to the cache.

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

Static Memory Allocation

1. Fast allocation and deallocation since it's done at compile-time.
2. No runtime overhead for memory management.

**Dynamic Memory Allocation (Heap Allocation)**

1. Flexible and suitable for dynamic workloads where memory needs change at runtime.
2. Efficient use of memory since it can be allocated and deallocated as needed.

Stack Allocation

1. Very fast allocation and deallocation as it involves simple pointer manipulation.
2. No fragmentation since memory is managed in a contiguous block.

**Memory Pools (Pool Allocation)**

1. Fast allocation and deallocation since it avoids the overhead of general-purpose allocators.
2. Can improve cache performance due to better locality of reference.

Custom Allocators

1. Can be optimized for specific usage patterns, leading to better performance.
2. Can reduce fragmentation and improve cache locality.

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

Containerization:

Challenges:

1. Ensuring containers don't interfere with each other's memory.
2. Frequent allocation/deallocation can fragment memory.
3. Long-running containers might leak memory.

Solutions:

1. Use cgroups to manage memory limits and isolation.
2. Use efficient memory allocation strategies.
3. Continuously monitor memory usage and leaks.

Real-Time Systems

Challenges:

1. Memory access times must be predictable.
2. Avoid fragmentation to ensure consistent performance.

Solutions:

1. Allocate memory in advance to avoid dynamic allocation.
2. Lock memory to prevent it from being swapped out.