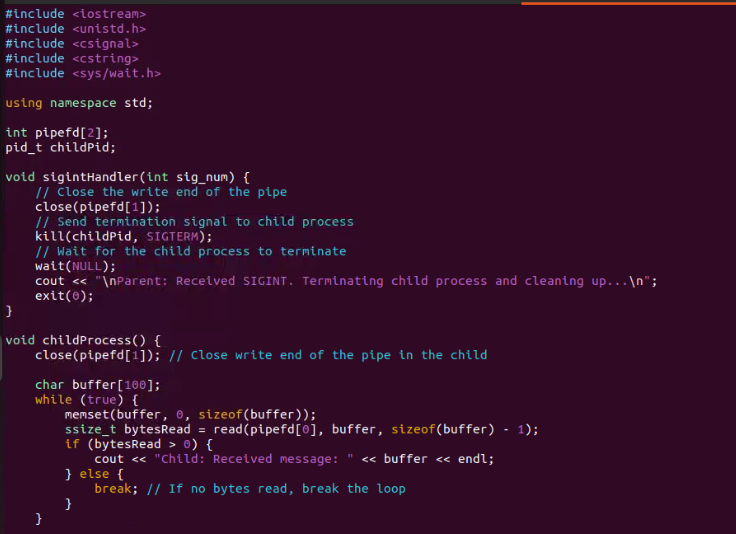
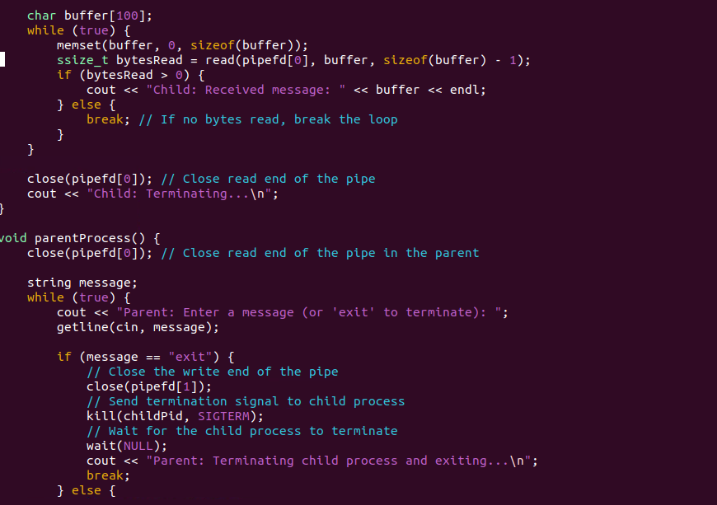
LINUX-31-7-24

1.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.

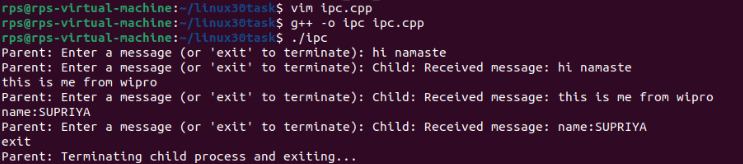
CODE:



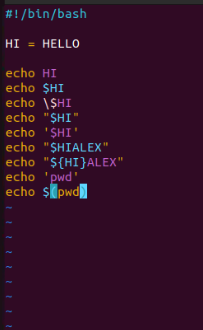




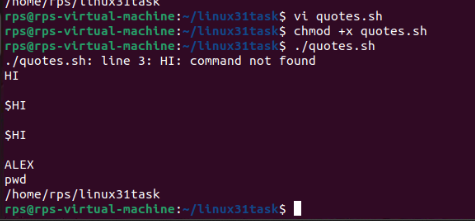
OUTPUT:



2.QUOTES SHELL SCRIPTING :

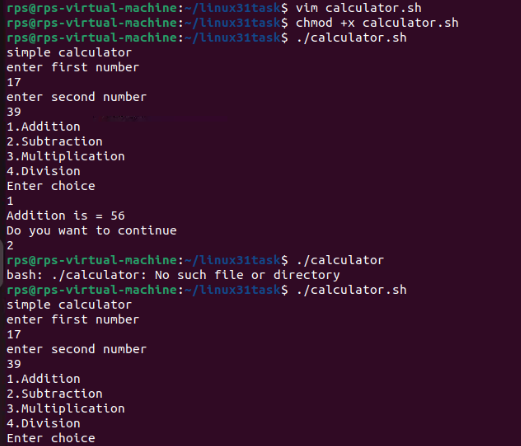


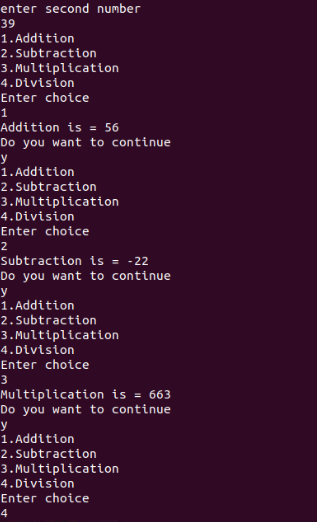
Output:

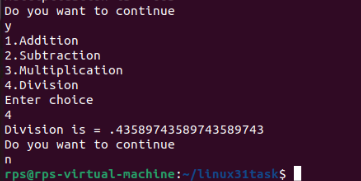


3.code that I have typed:











4. 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.

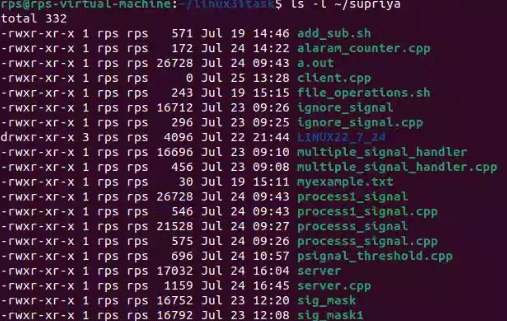
Change permissions of all files in the specified directory to rwxr-xr-x.

Print a message indicating the completion of the permission change.



Output:





5.Count Files and Directories

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

Instructions:

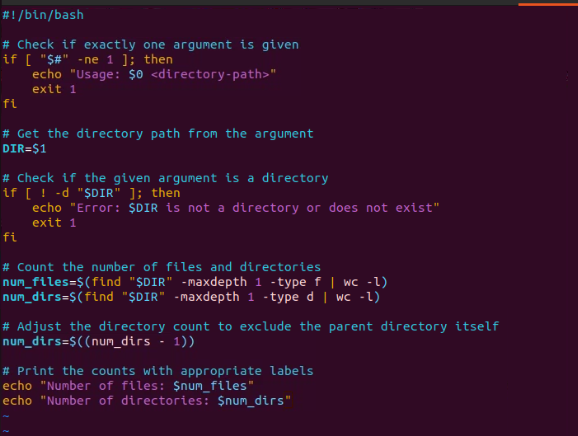
The script should accept one argument, the directory path.

Count the number of files and directories separately.

Print the counts with appropriate labels.

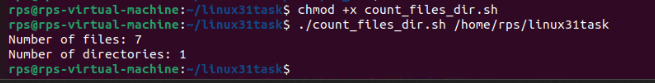
Sample Input:

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



Output:





6: 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.

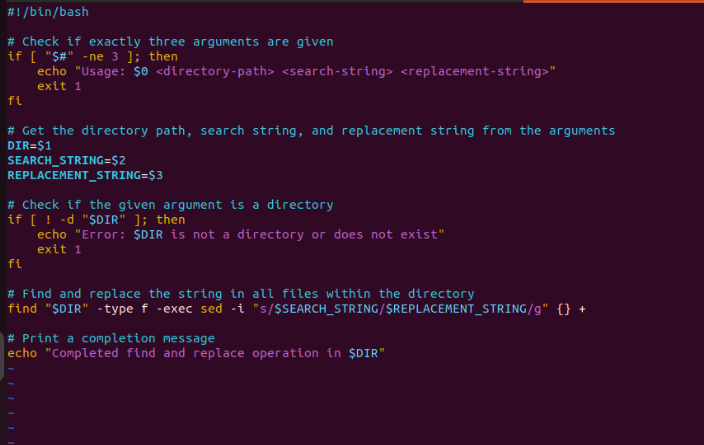
Search for the specified string in all files within the directory.

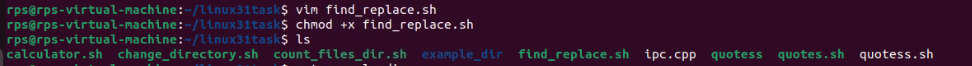
Replace the string with the given replacement string in all occurrences.

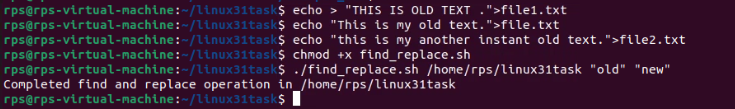
Print a message indicating the completion of the find and replace operation.

Sample Input:

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







7.

Problem 4: Disk Usage Report

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

Instructions:

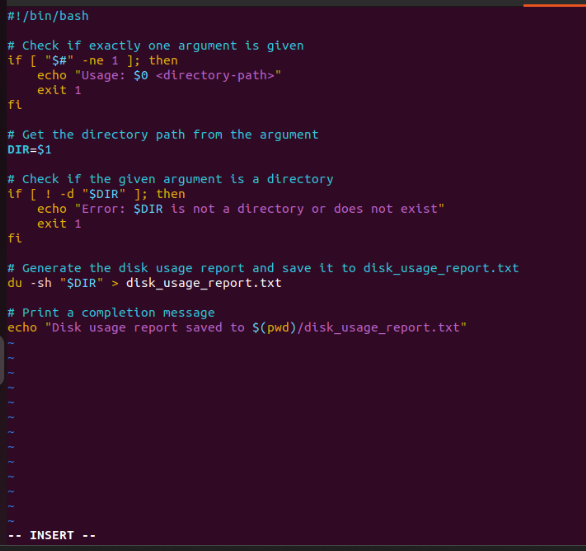
The script should accept one argument, the directory path.

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

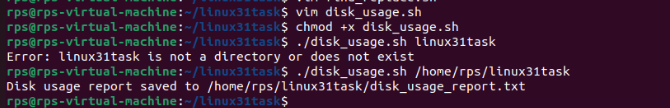
Save the report to a file named disk\_usage\_report.txt in the current directory.

Print a message indicating where the report is saved.

CODE:



Output:



8. 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:

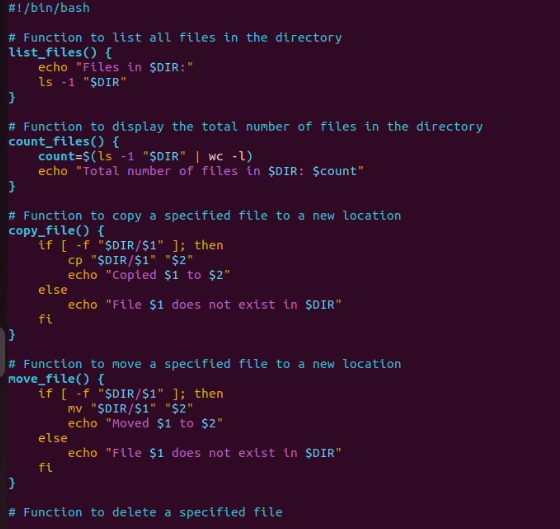
List all files in the directory.

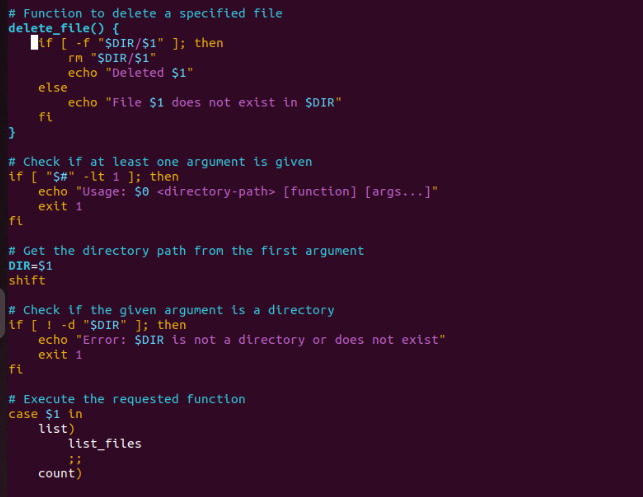
Display the total number of files.

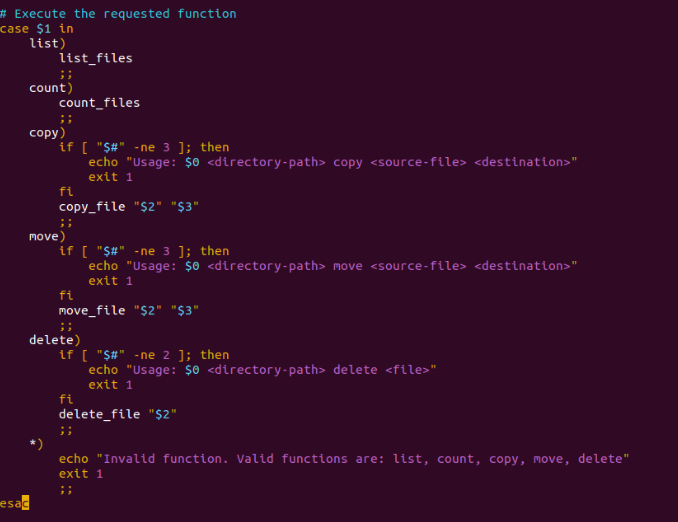
Copy a specified file to a new location.

Move a specified file to a new location.

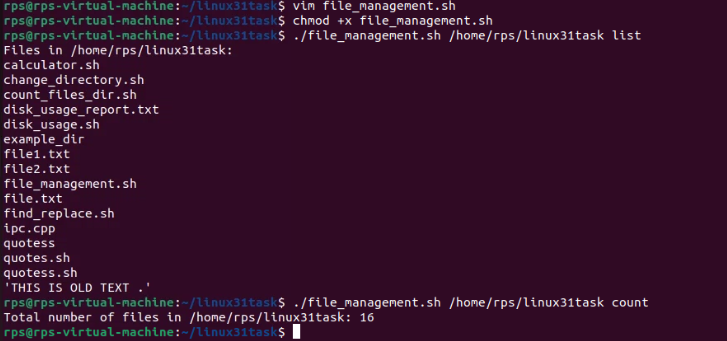
Delete a specified file.

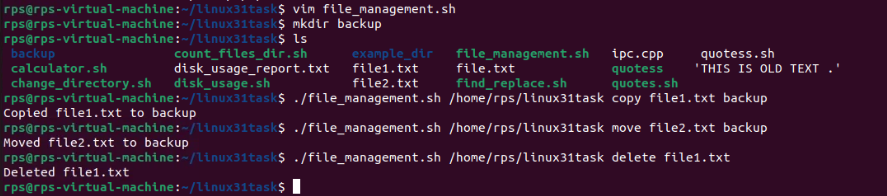






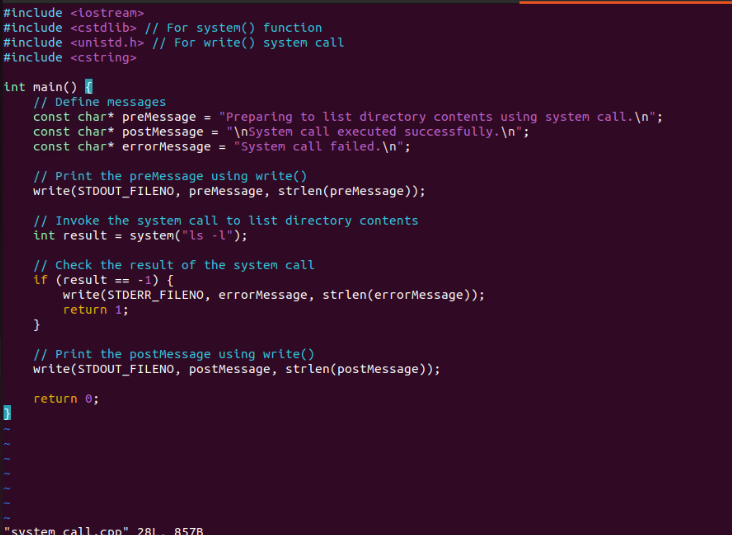
Output:



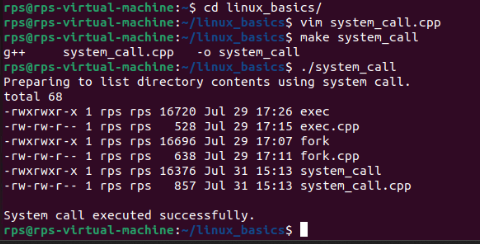


NEW TOPIC:

SYSTEMCALL : Code:A

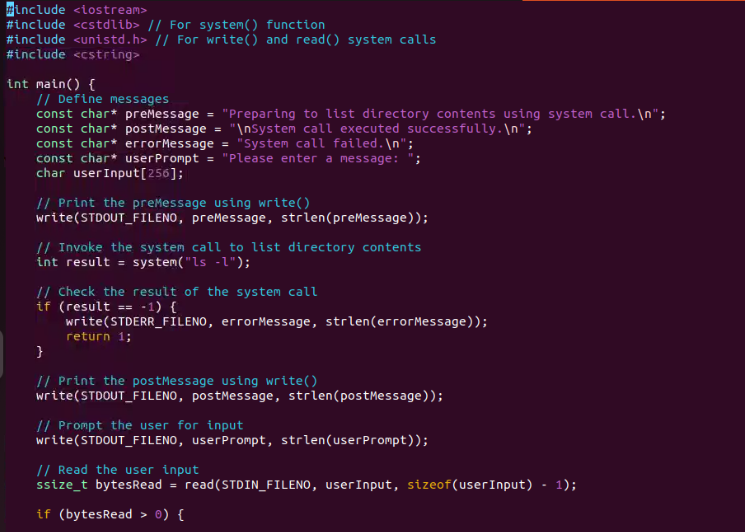


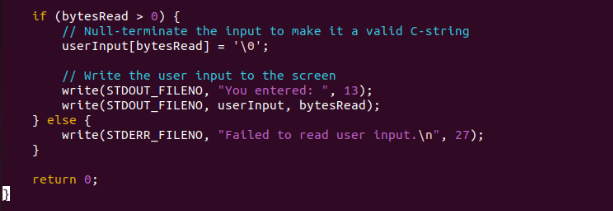
Output:



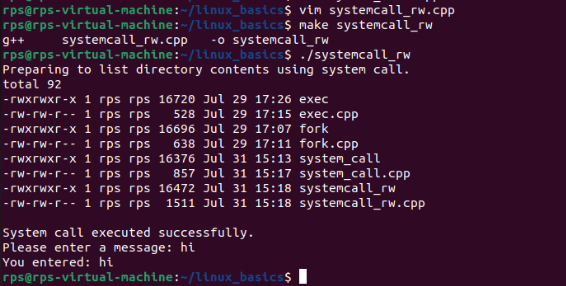
b.systemcall- read write

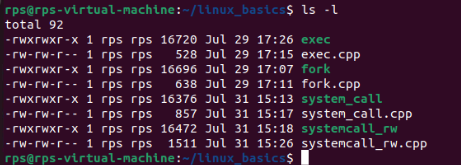
code:





output:





NEWTASK ON FILE-OPERATIONS:

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.

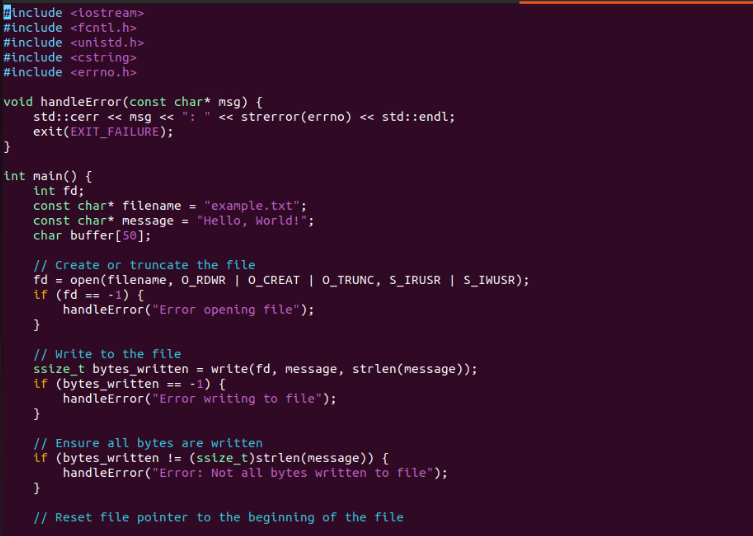
Delete the File:

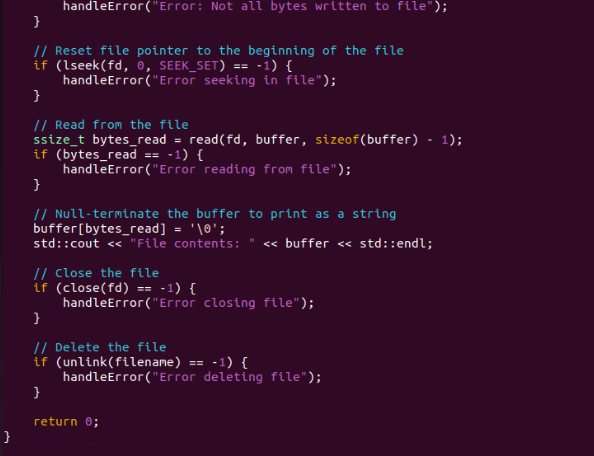
Close the file descriptor using the close system call.

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

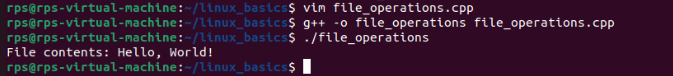
Error Handling:

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.





Output:



THEORY QUESTIONS:

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 a fundamental concept in modern operating systems, including Linux. It abstracts the physical memory of the system, providing each process with its own virtual address space. This abstraction offers several benefits:

1. **Isolation and Security**: Each process operates in its own virtual address space, which isolates processes from each other and increases security.
2. **Memory Management Flexibility**: The operating system can manage memory more efficiently by dynamically allocating physical memory to processes as needed.
3. **Address Space Expansion**: Virtual memory allows the system to use more memory than physically available by utilizing disk space (swap).

**System Calls for Managing Virtual Memory**

The Linux kernel provides several system calls to manage virtual memory for processes:

**brk and sbrk**

The brk and sbrk system calls are used to control the amount of memory allocated to the data segment of a process, commonly known as the heap.

* **brk**:
  + **Prototype**: int brk(void\* end\_data\_segment);
  + **Description**: Sets the end of the data segment to the value specified by end\_data\_segment.
  + **Return Value**: Returns 0 on success, -1 on failure.
* **sbrk**:
  + **Prototype**: void\* sbrk(intptr\_t increment);
  + **Description**: Increases the program's data space by increment bytes.
  + **Return Value**: Returns the previous end of the data segment on success, (void\*)-1 on failure.

**Usage**: These calls are typically used by memory allocation functions in standard libraries like malloc to manage heap memory.

**mmap and munmap**

The mmap system call maps files or devices into memory, providing a method for a process to map a region of virtual memory to a file or device.

* **mmap**:
  + **Prototype**: void\* mmap(void\* addr, size\_t length, int prot, int flags, int fd, off\_t offset);
  + **Parameters**:
    - addr: Starting address for the new mapping (can be NULL to let the kernel choose the address).
    - length: Length of the mapping.
    - prot: Desired memory protection of the mapping (e.g., PROT\_READ, PROT\_WRITE).
    - flags: Flags that determine the nature of the mapping (e.g., MAP\_SHARED, MAP\_PRIVATE).
    - fd: File descriptor of the file to be mapped.
    - offset: Offset in the file where the mapping starts.
  + **Return Value**: Returns a pointer to the mapped area on success, MAP\_FAILED on failure.
* **munmap**:
  + **Prototype**: int munmap(void\* addr, size\_t length);
  + **Description**: Unmaps a mapped region of memory.
  + **Return Value**: Returns 0 on success, -1 on failure.

**Usage**: These calls are used to map files into memory for efficient file I/O operations or to allocate memory for special purposes (e.g., shared memory).

**Implications of Overcommitting Memory**

Linux allows memory overcommitment, meaning it can allocate more memory to processes than is physically available, relying on the assumption that not all allocated memory will be used simultaneously. While this can improve memory utilization, it has risks:

* **Out of Memory (OOM)**: If processes use more memory than physically available, the system can run out of memory, causing the OOM killer to terminate processes to free up memory.
* **Performance Degradation**: Excessive swapping to disk can occur, significantly slowing down the system.

**Mechanisms to Handle Memory Pressure**

Linux employs several mechanisms to handle memory pressure:

* **OOM Killer**: When the system runs out of memory, the OOM killer selects and terminates processes to reclaim memory.
* **Swapping**: The kernel can move inactive pages to disk to free up physical memory.
* **Memory Reclamation**: The kernel periodically scans and reclaims unused or less frequently used memory pages.

**Further Exploration**

**Deep Dive into System Calls**

* **brk**:
  + Changes the end of the data segment, growing or shrinking the heap.
  + Used internally by memory allocators like malloc.
* **mmap**:
  + Used for memory-mapped file I/O, anonymous memory allocation, and shared memory.
  + Parameters like prot and flags determine the behavior and permissions of the mapped region.
* **munmap**:
  + Releases a mapped region of memory, freeing resources.

**Memory Allocation Algorithms**

* **Buddy System**: Divides memory into blocks of sizes that are powers of two. Efficient for allocations and deallocations but can suffer from fragmentation.
* **Slab Allocator**: Caches frequently used objects to reduce overhead and fragmentation, used for kernel memory allocation.

**Performance Implications**

Different memory management techniques can impact performance under various workloads. For example, excessive use of mmap can lead to fragmentation and performance degradation in certain scenarios.

**Memory Management in Specific Scenarios**

* **Containerization**: Containers use cgroups to limit memory usage and namespaces to provide isolated address spaces.
* **Real-Time Systems**: Require deterministic memory allocation with minimal latency, often avoiding techniques like swapping.

**Conclusion**

Virtual memory management in Linux is a complex but critical aspect that provides isolation, flexibility, and efficient memory usage. The kernel uses system calls like brk, mmap, and munmap to manage memory, with various strategies and mechanisms to handle memory pressure and ensure system stability. Understanding these concepts and their implications is crucial for optimizing performance and ensuring robust memory management in different scenarios.