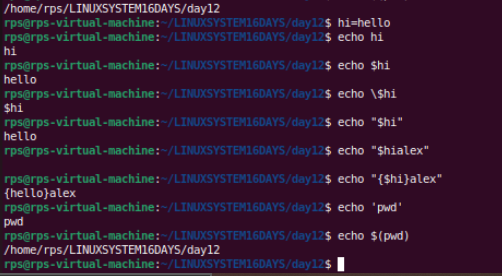
DAY12 BASH SCRIPTING |

ERROR HANDLING|

SYSTEM CALLS

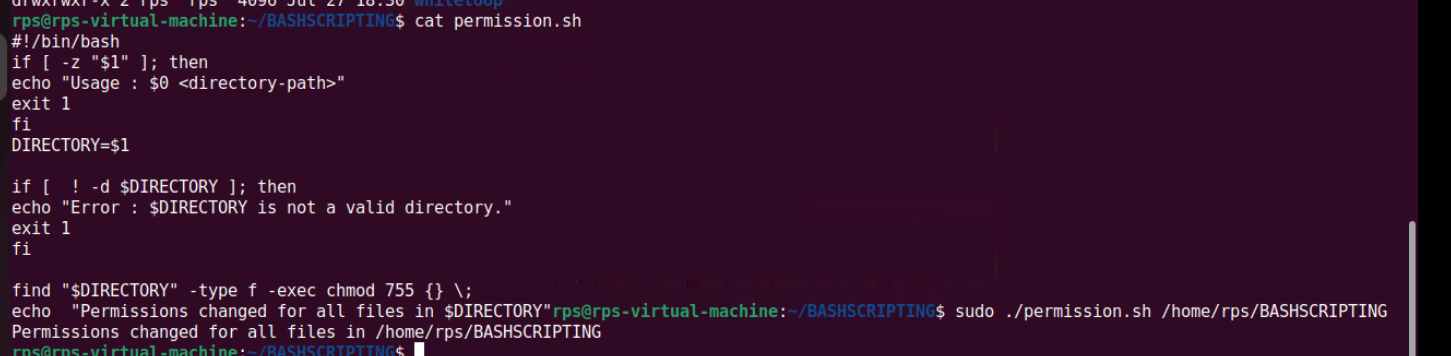
**Basic script command**



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 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.

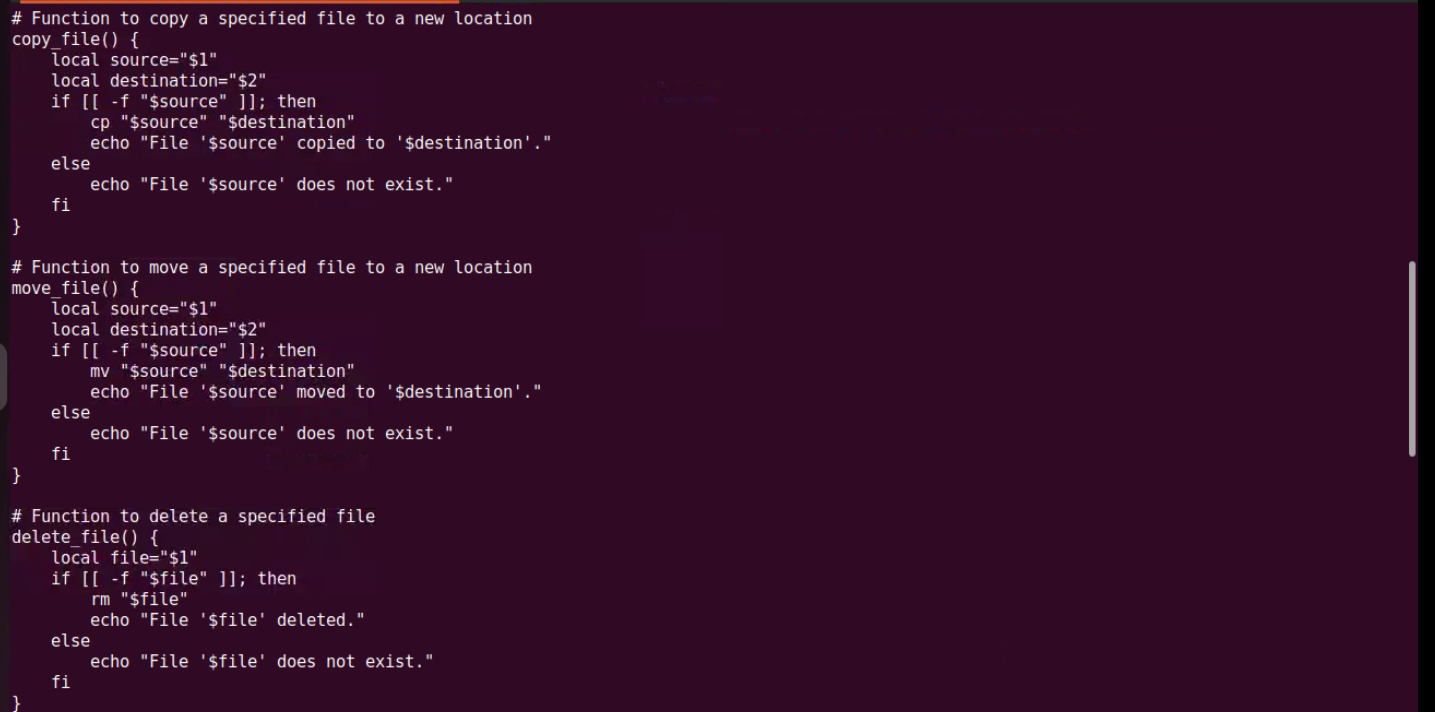
Display the total number of files.

Copy a specified file to a new location.

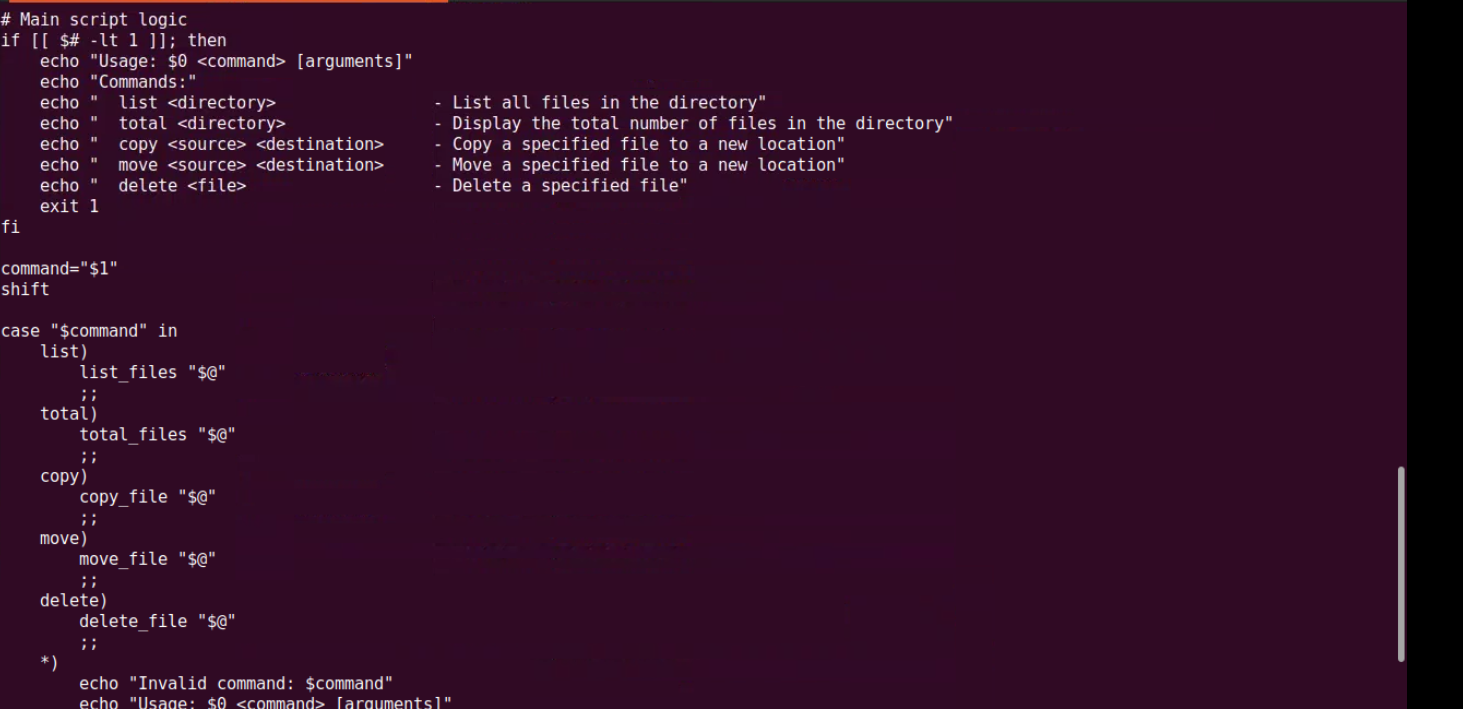
Move a specified file to a new location.

Delete a specified file.

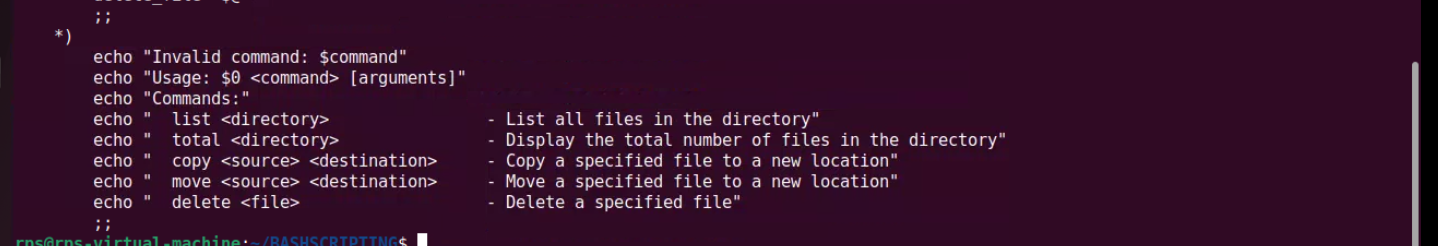
Part1



Part2



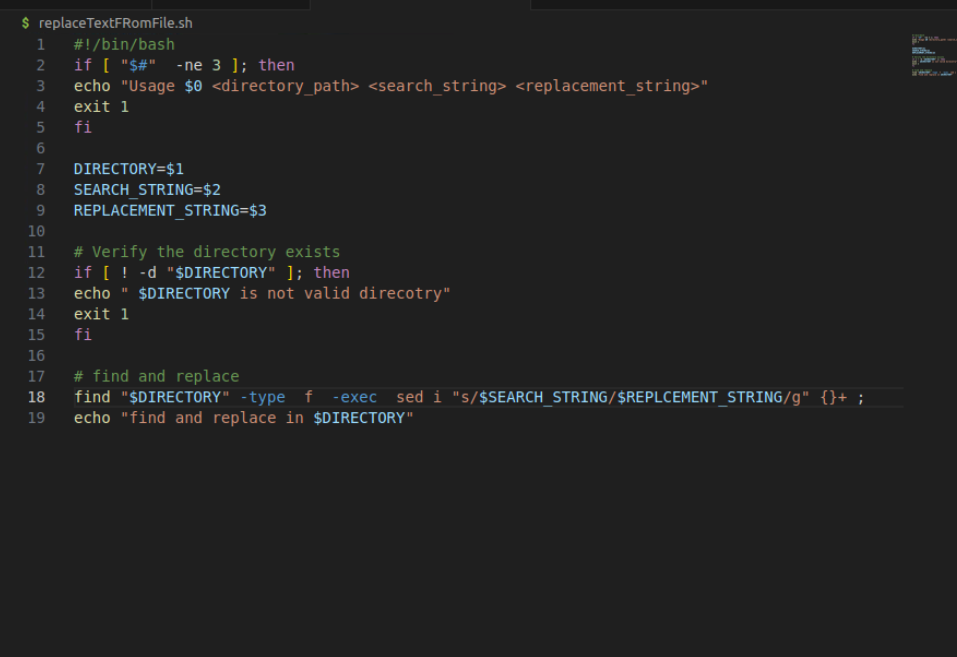
Part 3



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 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:

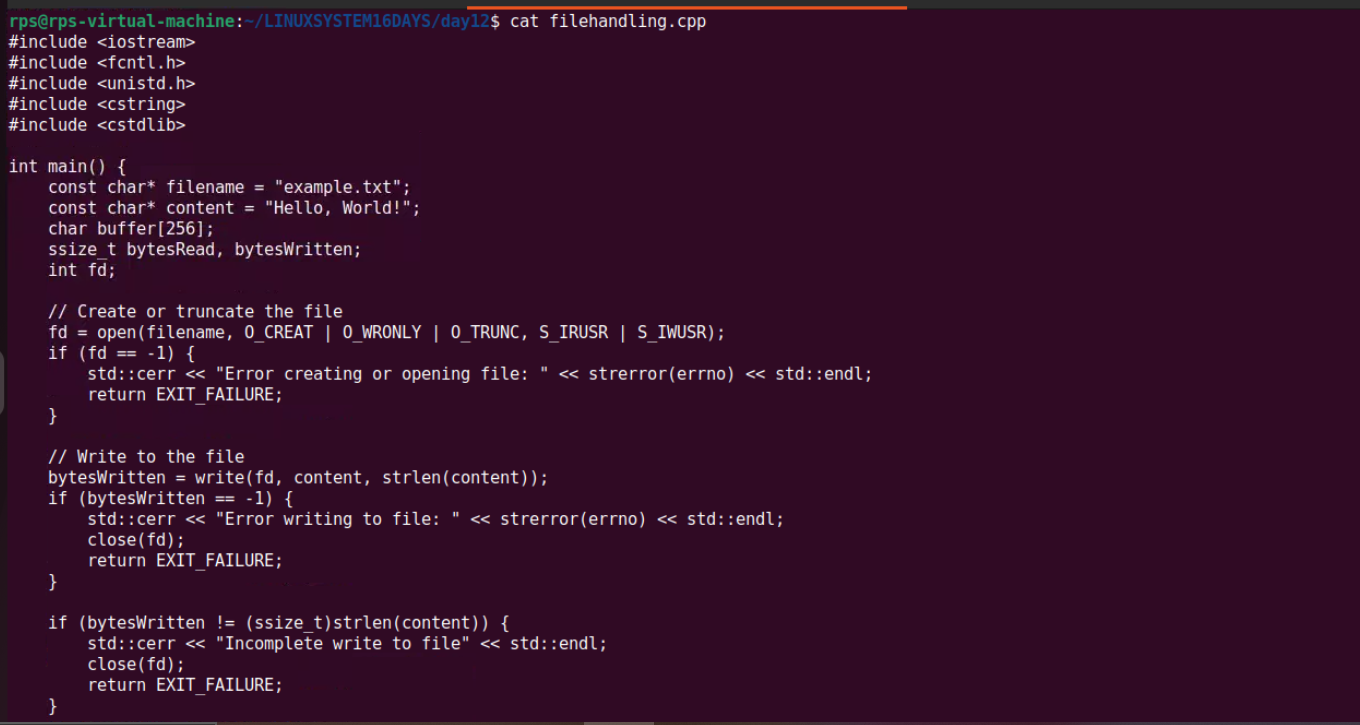
Use the open system call to create a new file named "example.txt" with read and write permissions.  
If the file already exists, truncate its contents.  
Write to the File:

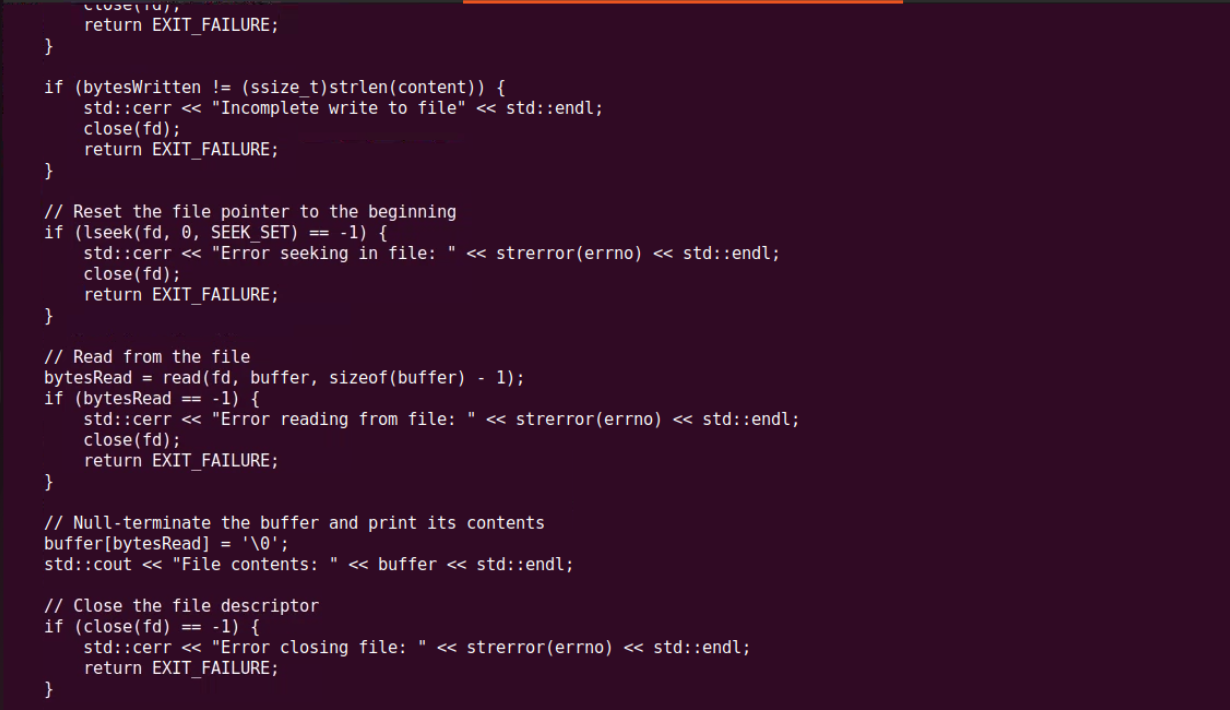
Write the string "Hello, World!" to the file using the write system call.  
Ensure that all bytes are written to the file.  
Read from the File:

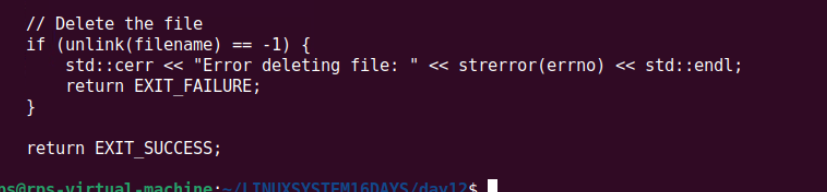
Use the lseek system call to reset the file pointer to the beginning of the file.  
Read the contents of the file using the read system call and store it in a buffer.  
Print the contents of the buffer to the standard output.  
Delete the File:

Close the file descriptor using the close system call.  
Use the unlink system call to delete the file "example.txt".  
Error Handling:

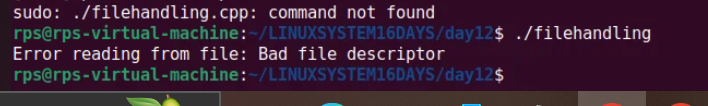
Ensure proper error handling for each system call. If a system call fails, print an error message and exit the program with a non-zero status.







OUTPUT



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.

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.

### **Virtual Memory in Linux Memory Management**

**Virtual Memory**

Virtual memory is a critical component of modern operating systems, including Linux. It provides an abstraction that allows processes to use memory in a more flexible and efficient way than physical memory alone. The kernel handles this abstraction by mapping virtual addresses used by processes to physical addresses in the computer's RAM.

**Key Advantages of Virtual Memory:**

* **Isolation:** Each process operates in its own virtual address space, preventing it from interfering with other processes.
* **Efficiency:** The system can use memory more flexibly, swapping out inactive pages to disk (swap space) and loading them back into RAM as needed.

### **System Calls for Managing Virtual Memory**

The kernel uses several system calls to manage virtual memory for processes, notably brk, mmap, and munmap.

**brk System Call**

* **Purpose:** Adjusts the end of the data segment (the program break) to allocate or deallocate memory.
* **Usage:** Typically used by the malloc library to allocate heap memory.
* **Parameters:**
  + brk(void \*addr) - Sets the end of the data segment to the value specified by addr.
* **Return Values:**
  + Returns 0 on success, or -1 on error.
* **Common Use Cases:**
  + Simple memory allocations for small programs.
  + Increasing or decreasing the size of the heap.

**mmap and munmap System Calls**

* **mmap:**
  + **Purpose:** Maps files or devices into memory, or allocates memory in the virtual address space.
  + **Usage:** Allows fine-grained control over memory regions and is used for memory-mapped file I/O.
  + **Parameters:**
    - void \*addr - Starting address for the new mapping (or NULL to let the kernel choose).
    - size\_t length - Length of the mapping.
    - int prot - Desired memory protection of the mapping (e.g., PROT\_READ, PROT\_WRITE).
    - int flags - Determines the nature of the mapping (e.g., MAP\_SHARED, MAP\_PRIVATE).
    - int fd - File descriptor (if mapping a file).
    - off\_t offset - Offset in the file.
  + **Return Values:**
    - Returns the starting address of the mapped area on success, or MAP\_FAILED on error.
  + **Common Use Cases:**
    - Memory-mapped file I/O.
    - Allocating large memory regions.
* **munmap:**
  + **Purpose:** Unmaps a previously mapped memory region.
  + **Usage:** Deallocates memory regions created by mmap.
  + **Parameters:**
    - void \*addr - Address of the mapped region.
    - size\_t length - Length of the mapping.
  + **Return Values:**
    - Returns 0 on success, or -1 on error.
  + **Common Use Cases:**
    - Cleaning up memory mappings.
    - Deallocating memory regions when they are no longer needed.

### **Overcommitting Memory**

**Overcommitment**

Overcommitting memory refers to the practice of allowing processes to allocate more memory than is physically available in the system. Linux allows overcommitting by default, relying on the assumption that not all allocated memory will be used simultaneously.

**Implications of Overcommitting:**

* **Benefits:**
  + Improves system utilization by allowing more processes to run simultaneously.
  + Enables efficient use of memory resources.
* **Drawbacks:**
  + Risk of running out of memory if too many processes use their allocated memory simultaneously.
  + May lead to the Out-Of-Memory (OOM) killer being invoked to terminate processes and free up memory.

**Handling Memory Pressure:**

When the system experiences memory pressure, Linux employs several mechanisms:

* **Swapping:** Moves inactive pages from RAM to swap space, freeing up physical memory.
* **OOM Killer:** If memory pressure becomes critical, the OOM killer terminates processes to reclaim memory. It uses heuristics to select processes that are likely to free the most memory and have the least impact on system functionality.

### **Further Exploration**

**Deep Dive into Specific System Calls:**

* **brk:**
  + Used for simple heap memory management.
  + Limited by the need for contiguous memory regions.
* **mmap and munmap:**
  + Provide more flexible and powerful memory management.
  + Enable memory-mapped file I/O, which can improve performance for large files by reducing the number of system calls.

**Memory Allocation Algorithms:**

* **Buddy System:**
  + Splits memory into blocks of size 2^n.
  + Efficiently handles memory fragmentation but can suffer from internal fragmentation.
* **Slab Allocator:**
  + Manages caches of objects for frequently used kernel data structures.
  + Reduces fragmentation and improves performance for small object allocations.