Day – 13 [LSP Assignment]

1. Question:

1. 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 over committing memory and the mechanisms Linux employs to handle memory pressure ?

Role of Virtual Memory in Linux Memory Management

Virtual memory is a crucial component of Linux memory management that enables the kernel to manage memory efficiently and effectively. It allows multiple processes to share the same physical memory space while maintaining the illusion of separate memory spaces for each process. The kernel uses virtual memory to:

1. The kernel translates virtual addresses used by processes to physical addresses in RAM.
2. Virtual memory ensures that processes cannot access each other's memory spaces, preventing data corruption and security breaches.
3. Virtual memory allows multiple processes to share the same physical memory pages, reducing memory usage and improving performance.
4. The kernel manages memory allocation and deallocation requests from processes using system calls like brk, mmap, and munmap.

System Calls for Virtual Memory Management

The kernel uses the following system calls to manage virtual memory for processes:

1. brk: Changes the program break, which is the end of the data segment. Used to allocate or de-allocate memory for the heap.
2. mmap: Maps a file or device into memory, creating a new virtual memory mapping.Used to allocate memory for shared libraries, files, or devices.
3. munmap: Unmaps a virtual memory mapping, releasing the associated memory pages. Used to de-allocate memory previously allocated using mmap.

Implications of Over Committing Memory

Over committing memory occurs when the kernel allocates more memory than is physically available. This can lead to:

1. Memory pressure: The kernel must reclaim memory from other processes or swap memory to disk, causing performance degradation.
2. OOM (Out of Memory) killer: The kernel terminates processes to free up memory, potentially leading to data loss or corruption.

Mechanisms to Handle Memory Pressure

Linux employs several mechanisms to handle memory pressure:

1. Swap space: The kernel swaps less frequently used memory pages to disk, freeing up physical memory.
2. OOM killer: Terminates processes to free up memory.
3. Memory reclaim: The kernel reclaims memory from caches, buffers, and other sources.
4. Memory compaction: The kernel defragments memory to reduce fragmentation and improve memory allocation efficiency.

2. Potential Areas for Further Exploration:

1. 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?

Specific system calls

i. brk:

- Parameters: brk(void *addr)

- Return value: int (0 on success, -1 on error)

- Common use cases: Allocate or deallocate memory for the heap

- Inner workings: brk changes the program break, which is the end of the data segment. It increments or decrements the break value to allocate or deallocate memory.

ii. mmap:

- Parameters: mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)

- Return value: void * (mapped memory address on success, MAP_FAILED on error)

- Common use cases: Map a file or device into memory, create a new virtual memory mapping

- Inner workings: mmap creates a new virtual memory mapping by allocating a range of virtual addresses and mapping them to a file or device.

iii. munmap:

- Parameters: munmap(void *addr, size_t length)

- Return value: int (0 on success, -1 on error)

- Common use cases: Unmap a virtual memory mapping, release associated memory pages

- Inner workings: munmap releases the virtual memory mapping and associated memory pages, making them available for other uses.

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

   Memory allocation algorithms

   - Buddy System:

- Allocates memory in powers of 2 (2^n)

- Splits large blocks into smaller ones (buddies) to satisfy allocation requests

- Merges adjacent free blocks to reduce fragmentation

   - Slab Allocator:

- Allocates memory in fixed-size blocks (slabs)

- Uses a cache to store frequently allocated objects

- Reduces fragmentation by allocating objects of the same size together

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

- **Memory allocation overhead:** Measuring the time and resources spent on memory allocation and deallocation

- **Memory fragmentation:** Analyzing the impact of fragmentation on memory allocation performance

- **Cache performance:** Evaluating the effect of memory allocation on cache hit rates and performance

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

   Memory management in specific scenarios

- Containerization:

- Memory management challenges: Limited memory resources, isolation between containers

- Solutions: Using cgroups to limit memory usage, implementing memory-aware container scheduling

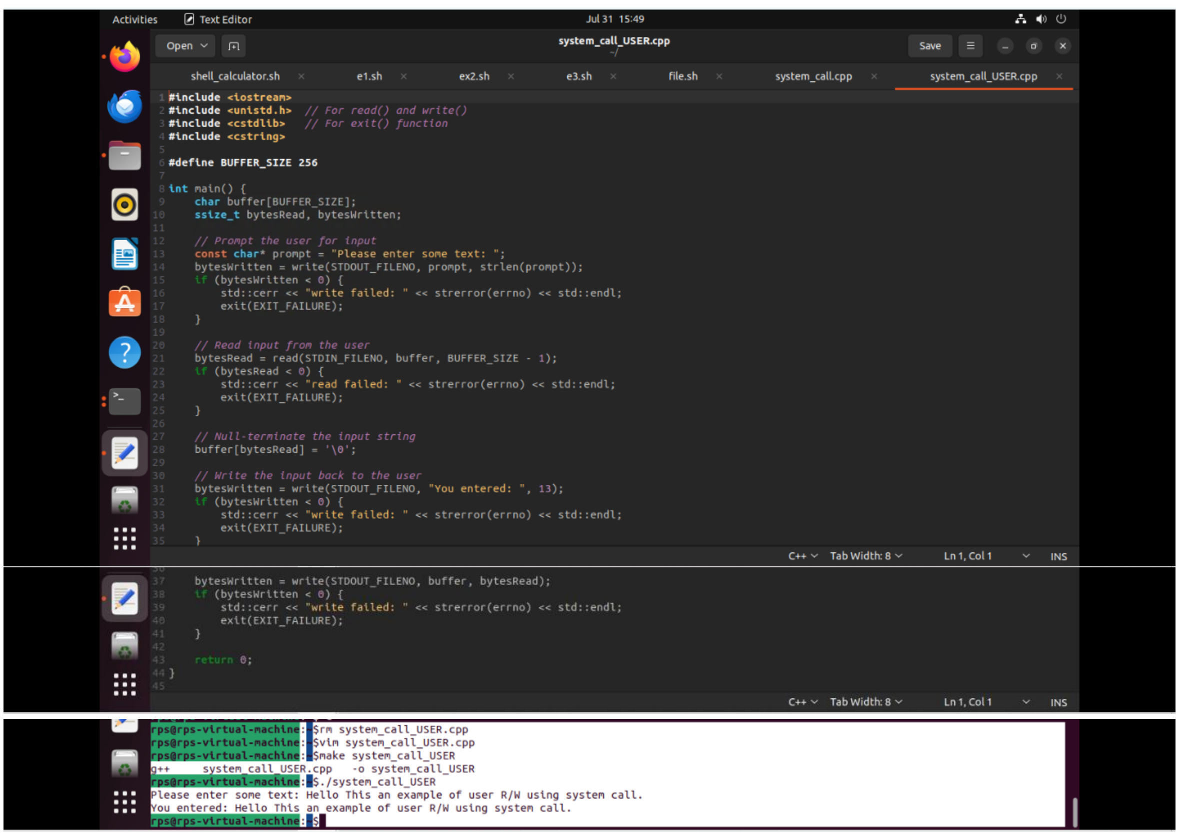
- Real-time systems:

- Memory management challenges: Predictable and fast memory allocation, avoiding page faults

- Solutions: Using real-time memory allocation algorithms, implementing memory locking and pinning.

2. Codes on System API’s

1. **Problem Statement:** Write the code please read from user and write on screen using read and write API’s in cpp using system calls.



2. **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:

a. **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.

b. **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.

c. **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.

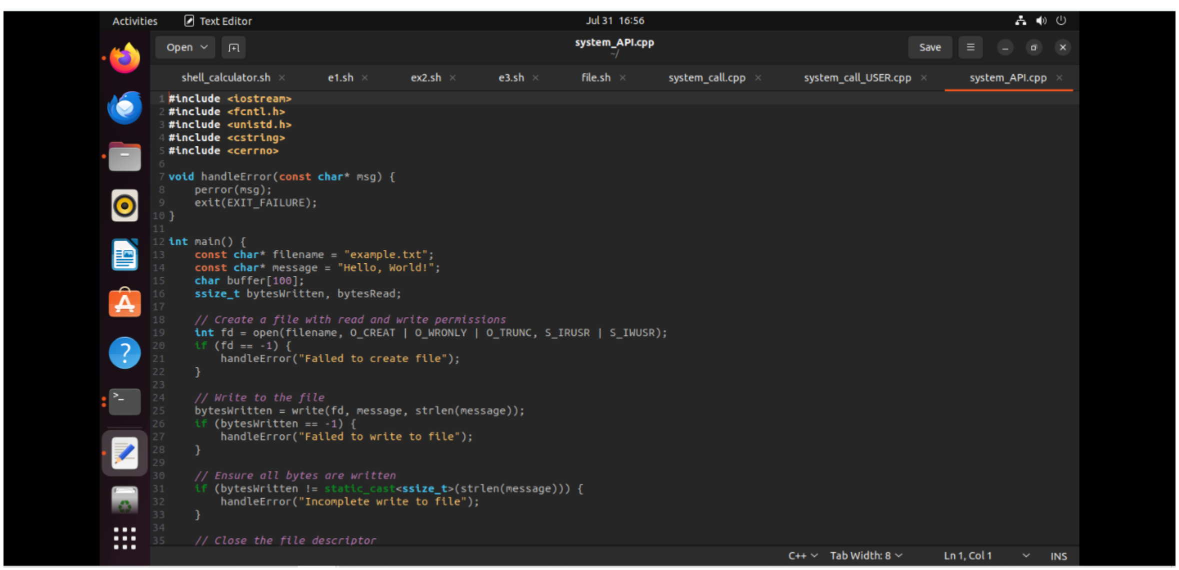
d. **Delete the File:**

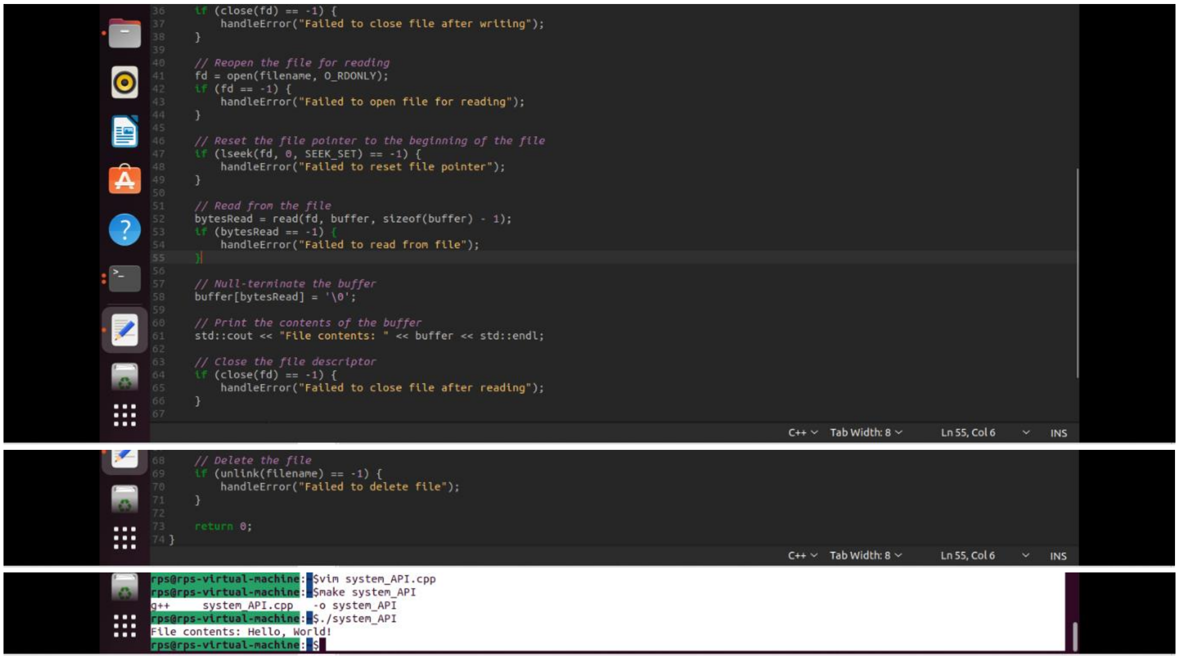
Close the file descriptor using the close system call.

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

e. **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.





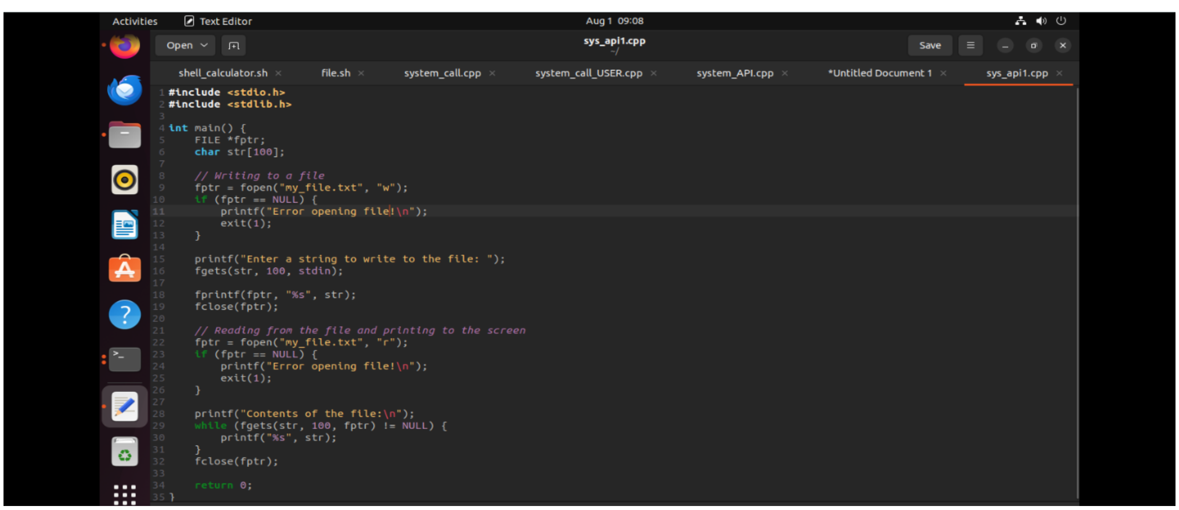
3. Problem Statement : Write a C and C++ program that performs the following tasks:

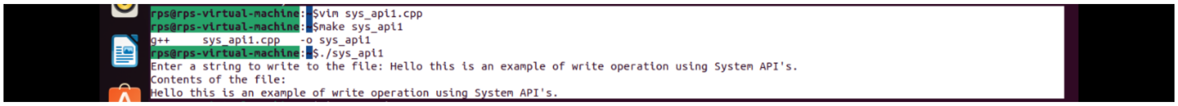
1. Prompts the user to enter a string and writes this string to a file named `my_file.txt`.
2. Reads the contents of `my_file.txt` and prints it to the screen.

The program should include the necessary error handling to ensure that:

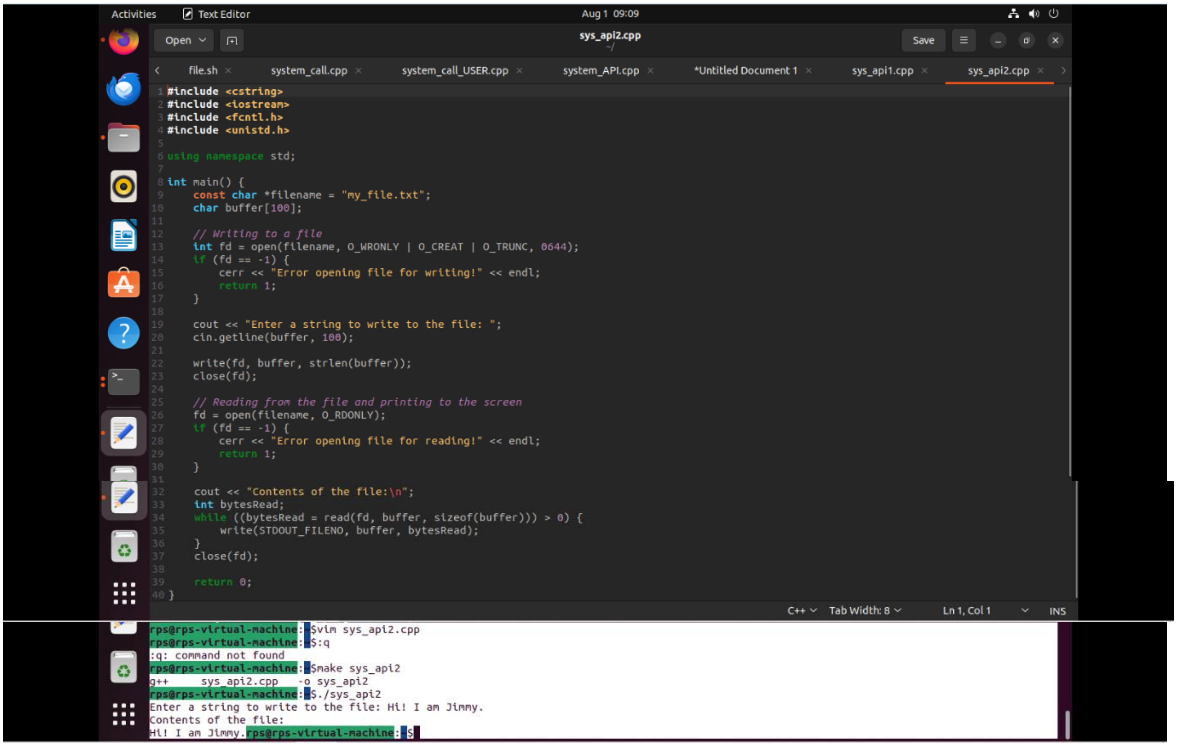
- The file is successfully opened for writing.
- The string is successfully written to the file.
- The file is successfully opened for reading.
- The contents of the file are correctly read and displayed.

A. C code

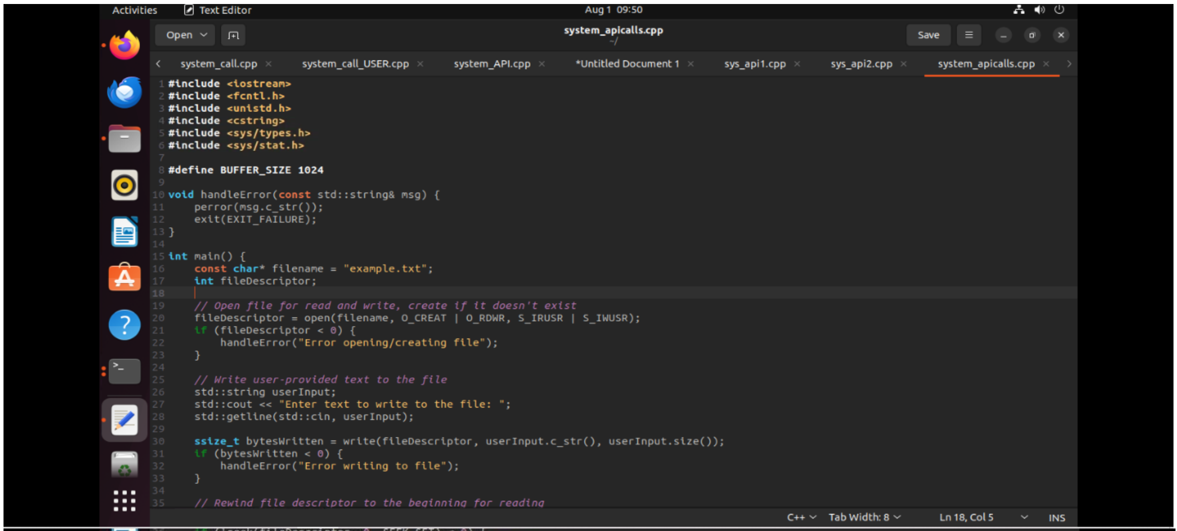




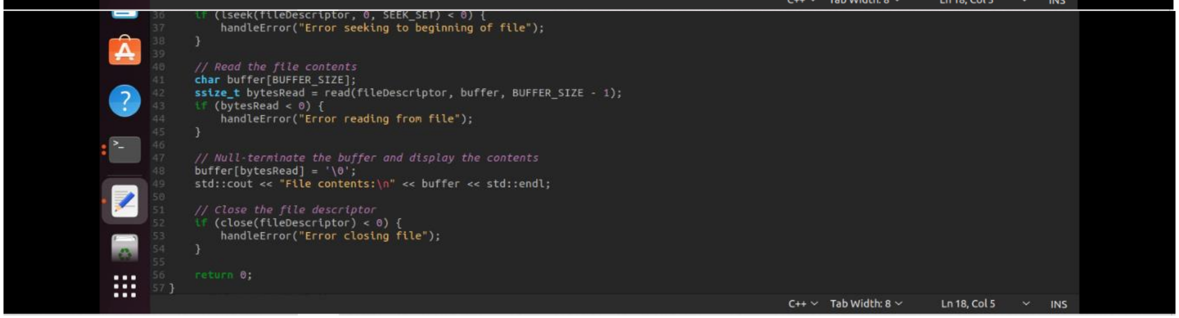
B. C++ code



4. **Problem Statement: 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.




```cpp
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include <cstring>
#include <sys/types.h>
#include <sys/stat.h>

#define BUFFER_SIZE 1024

void handleError(const std::string& msg) {
    perror(msg.c_str());
    exit(EXIT_FAILURE);
}

int main() {
    const char* filename = "example.txt";
    int fileDescriptor;

    // Open file for read and write, create if it doesn't exist
    fileDescriptor = open(filename, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    if (fileDescriptor < 0) {
        handleError("Error opening/creating file");
    }

    // Write user-provided text to the file
    std::string userInput;
    std::cout << "Enter text to write to the file: ";
    std::getline(std::cin, userInput);

    ssize_t bytesWritten = write(fileDescriptor, userInput.c_str(), userInput.size());
    if (bytesWritten < 0) {
        handleError("Error writing to file");
    }

    // Rewind file descriptor to the beginning for reading
```




```cpp
    if (lseek(fileDescriptor, 0, SEEK_SET) < 0) {
        handleError("Error seeking to beginning of file");
    }

    // Read the file contents
    char buffer[BUFFER_SIZE];
    ssize_t bytesRead = read(fileDescriptor, buffer, BUFFER_SIZE - 1);
    if (bytesRead < 0) {
        handleError("Error reading from file");
    }

    // Null-terminate the buffer and display the contents
    buffer[bytesRead] = '\0';
    std::cout << "File contents:\n" << buffer << std::endl;

    // Close the file descriptor
    if (close(fileDescriptor) < 0) {
        handleError("Error closing file");
    }

    return 0;
}
```