Assignment Report: CSC3150 Assignment1

Name: Wang Jiaju - Student ID:121090544

Your report should follow the template with the following section structure.

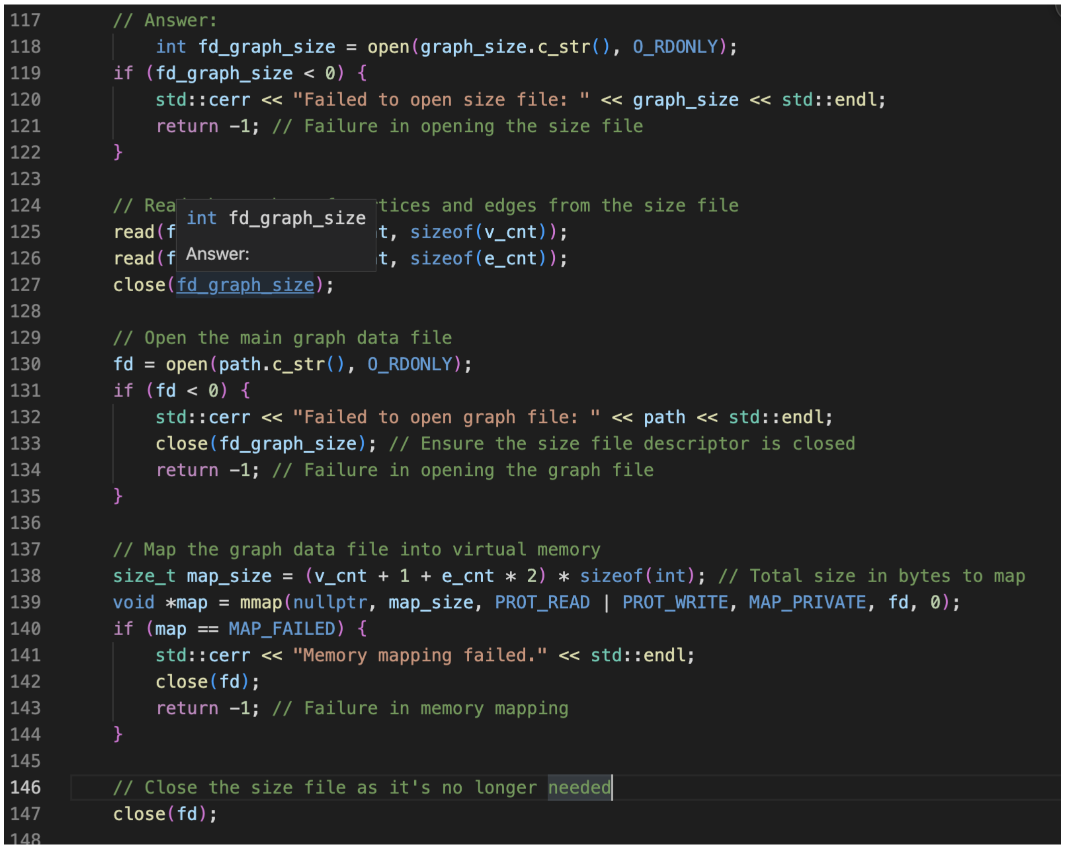
No page limitation

1 Introduction [2']

This report delves into Assignment 1, focusing on leveraging system calls for effective file and process management within a programming context. The assignment's essence lies in the practical application of system calls to manage files, map files into memory, and control processes, illustrating the foundational concepts of operating system functionalities. Through the completion of this assignment, the goal was to acquire a hands-on understanding of system-level programming, specifically how system calls operate in managing files, memory mapping, and process control, thereby bridging the gap between user-space programs and the kernel.

2 Design [5']

The program was designed with a modular approach, segregating functionalities into distinct segments for manageability and clarity. The file management module leverages system calls such as open(), read(), and write() to interact with files at a low level, ensuring efficient data handling without the overhead of higher-level abstractions. For memory mapping, the mmap() system call was utilized to directly map files into the process's address space, facilitating faster and more efficient file manipulation. Process control was managed using the fork(), wait(), and related calls to create and synchronize processes, crucial for concurrent execution and resource management. The design prioritizes performance and resource efficiency, aiming to minimize overhead and maximize throughput in operations. This C++ code is part of a graph processing application that deals with loading, mapping, and managing graph data in memory, as well as writing computation results back to a file. The graph data is structured and stored in a binary format to optimize performance and memory usage, particularly useful for large datasets. Let's break down each part of the code to understand its functionality: Task 1.1: Load Application Data into Memory Objective: Load graph data from binary files into memory structures for further computation. This is intended for smaller datasets where loading the entire graph into RAM is feasible. Key Operations: The graph's size information (number of nodes and relations) is read from a '.size' file. Memory is dynamically allocated for vertices, edges, and their weights based on the read sizes. The main graph data, including node edge prefixes, relations, and weights, is loaded from a binary file. The code performs basic sanity checks to ensure that the loaded graph structure is valid (e.g., vertex prefixes are in increasing order and within bounds). Task 1.2: Map Application Data into Virtual Memory Objective: For larger datasets, this task maps the graph data file into virtual memory space instead of loading it all into RAM. This approach leverages memory-mapped files to access large files efficiently without requiring large amounts of RAM. Key Operations: Similar to the loading task, it starts by reading the graph's size information. It then opens the graph data file and maps it into the process's virtual address space using 'mmap'. This allows the application to treat the file contents as if they were loaded into memory, with the operating system handling the actual data loading on demand. Pointers for vertices, edges, and weights are set to the appropriate locations within the mapped memory space. The function returns the file descriptor for further operations like unmapping. Unmapping Memory Objective: Correctly release the mapped memory space and reset pointers when the graph data is no longer needed. Key Operations: The 'munmap' system call is used to unmap the previously mapped memory space. The file descriptor is closed, and the internal pointers are set to



```
    // Answer:
        int fd_graph_size = open(graph_size.c_str(), O_RDONLY);
    if (fd_graph_size < 0) {
        std::cerr << "Failed to open size file: " << graph_size << std::endl;
        return -1; // Failure in opening the size file
    }

    // Rea[illegible]tices and edges from the size file
    read(f[illegible]t, sizeof(v_cnt));
    read(f[illegible]t, sizeof(e_cnt));
    close(fd_graph_size);

    // Open the main graph data file
    fd = open(path.c_str(), O_RDONLY);
    if (fd < 0) {
        std::cerr << "Failed to open graph file: " << path << std::endl;
        close(fd_graph_size); // Ensure the size file descriptor is closed
        return -1; // Failure in opening the graph file
    }

    // Map the graph data file into virtual memory
    size_t map_size = (v_cnt + 1 + e_cnt * 2) * sizeof(int); // Total size in bytes to map
    void *map = mmap(nullptr, map_size, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
    if (map == MAP_FAILED) {
        std::cerr << "Memory mapping failed." << std::endl;
        close(fd);
        return -1; // Failure in memory mapping
    }

    // Close the size file as it's no longer needed
    close(fd);
```

Figure 1:

'nullptr' to prevent dangling references. Writing Application Answer Back to File Objective: Write computation results back to a binary file. Key Operations: The function takes an array of integers ('data') and its length ('len'), and writes this data to a specified file in binary format. This is useful for saving the results of graph computations (like shortest paths, centrality measures, etc.) back to disk. Summary This code demonstrates efficient techniques for handling large graph datasets in C++, including dynamic memory allocation, memory-mapped files, and binary file I/O. Such approaches are essential for high-performance computing applications where data size can exceed available RAM, and where speed and efficiency are critical. For Task1.2:just the Figure1 for Task2: Parent Process Logic: 1. Open IPC Channel: Opens the IPC file descriptor for read/write operations. 2. Memory Mapping: Maps a shared memory segment for IPC. 3. Busy Waiting: Waits for the child process to reset the control variable, indicating it's the parent's turn to perform its operation. 4. Operation Execution: Executes the 'produce' operation, manipulating shared data as needed. 5. Signal Completion: Updates the control variable to signal the child process that the parent has completed its task. 6.Cleanup: Unmaps the shared memory and closes the file descriptor. Child Process Logic: 1.Open IPC Channel: Similar to the parent process, it opens the IPC file for read/write operations. 2.Memory Mapping**: Maps the same shared memory segment for IPC. 3. Busy Waiting**: Waits for the parent process to complete its operation and give control to the child process. 4.Operation Execution: Performs the 'consume' operation, manipulating shared data as needed. 5.Reset Control Variable**: Resets the control variable to allow the parent process to proceed in the next iteration. 6.Cleanup: Unmaps the shared memory and closes the file descriptor. Key Points: IPC Mechanism: Utilizes shared memory for inter-process communication, enabling synchronization between parent and child processes. Control Variable: A simple integer at the beginning of the shared memory segment acts as a semaphore for coordinating operations. Busy Waiting: Both processes engage in busy waiting ('while' loops) for synchronization. This method is straightforward but can be inefficient if waiting periods are long because it consumes CPU time. Resource Management: Properly maps and unmaps memory and handles file descriptors to avoid leaks or dangling resources. Recommendations: Efficiency: For real-world applications, consider more efficient IPC mechanisms or synchronization primitives that avoid busy waiting, such as semaphores or condition variables, to reduce CPU usage. Error Handling: Implement comprehensive error handling around system calls (‘open‘, ‘mmap‘, etc.) to ensure robustness. Security: Ensure that shared resources (like the IPC file) are securely managed to prevent unauthorized access or resource conflicts. This setup, as outlined in your code, provides a foundational framework for a producer-consumer scenario between parent and child processes, demonstrating basic IPC and process synchronization principles.

3 Environment and Execution [2’]

The program was developed and tested on a Linux environment, capitalizing on the POSIX API for system calls, ensuring broad compatibility and performance. Execution requires a C++ compiler.Successful execution is evidenced by the program’s ability to read, process, and write binary files accurately, adhering to the assignment’s specifications for file management, memory mapping, and process control. Demonstrations of the program’s correct operation include screenshots or logs of terminal outputs, showing the expected results and successful completion messages.

4 Conclusion [2’]

Through the completion of Assignment 1, a deepened understanding of system calls and their pivotal role in system-level programming was achieved. The assignment illuminated the intricacies of file and process management, highlighting the efficiency and control offered by direct system call usage. It fostered an appreciation for the underlying mechanisms of operating systems and the critical balance between user-space applications and kernel-space operations. This experience has not only enhanced technical skills in system-level programming but also reinforced the importance of efficiency, design, and understanding of the operating system’s core functionalities.