RV Educational Institutions ®

RV College of Engineering ®

Autonomous Institution
Affiliated to Visvesvaraya
Technological University,
Belagavi

Approved by AICTE,
New Delhi

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS - CS235AI

REPORT

Submitted by

Varun A	**1RV22CS226**
Yatharth Yadav	**1RV22CS239**
T Vinay	**1RV22CS215**

Computer Science and Engineering
2023-2024

Synopsis should contain 3 paragraphs

1. Introduction
2. System Architecture
3. Methodology
4. Systems calls used
5. Output/results
6. Conclusion

Introduction

In the realm of modern computing systems, efficient communication mechanisms play a pivotal role in facilitating seamless interaction between various software components. Shared memory message queues represent a cornerstone in this landscape, offering fast and efficient inter-process communication (IPC) within a system. By leveraging shared memory regions and a message passing paradigm, these queues enable processes to exchange data swiftly, bypassing the overhead associated with traditional IPC mechanisms like sockets or pipes.

Designing a Shared Memory Message Queue via System Call Interface encapsulates the essence of crafting a robust and efficient communication toolset for contemporary computing environments. By delving into the intricacies of system call interfaces, this endeavor aims to provide developers with a comprehensive understanding of how to architect, implement, and optimize shared memory message queues within their software systems.

This introductory discourse sets the stage for exploring the foundational concepts underlying shared memory message queues and underscores the significance of employing system calls as the interface for interaction with the underlying operating system. Through a blend of theoretical insights and practical considerations, this endeavor endeavors to equip software engineers with the knowledge and techniques necessary to design and deploy shared memory message queues effectively, thereby empowering them to enhance the performance, scalability, and reliability of their applications.

Throughout this journey, we'll navigate the fundamental principles of inter-process communication, delve into the mechanics of shared memory management, dissect the intricacies of system call invocation, and culminate in the design and implementation of a robust shared memory message queue system. By embracing these principles and practices, developers can unlock new avenues for optimizing communication between processes, thereby fostering the development of faster, more responsive, and more resilient software systems.

System Architecture

Ubuntu:

Linux Kernel: Ubuntu, like many other Linux distributions, relies on the Linux kernel as its core component. The Linux kernel provides essential functionalities such as process management, memory management, device drivers, filesystem support, and networking capabilities. It serves as the interface between the hardware and the software layers of the system.

POSIX Compliance: Ubuntu adheres to POSIX standards, which define a set of operating system interfaces for compatibility across Unix-like systems. POSIX compliance ensures that Ubuntu supports a common set of APIs and utilities for inter-process communication, file operations, process management, and more, facilitating portability and interoperability with other POSIX-compliant systems.

Windows:

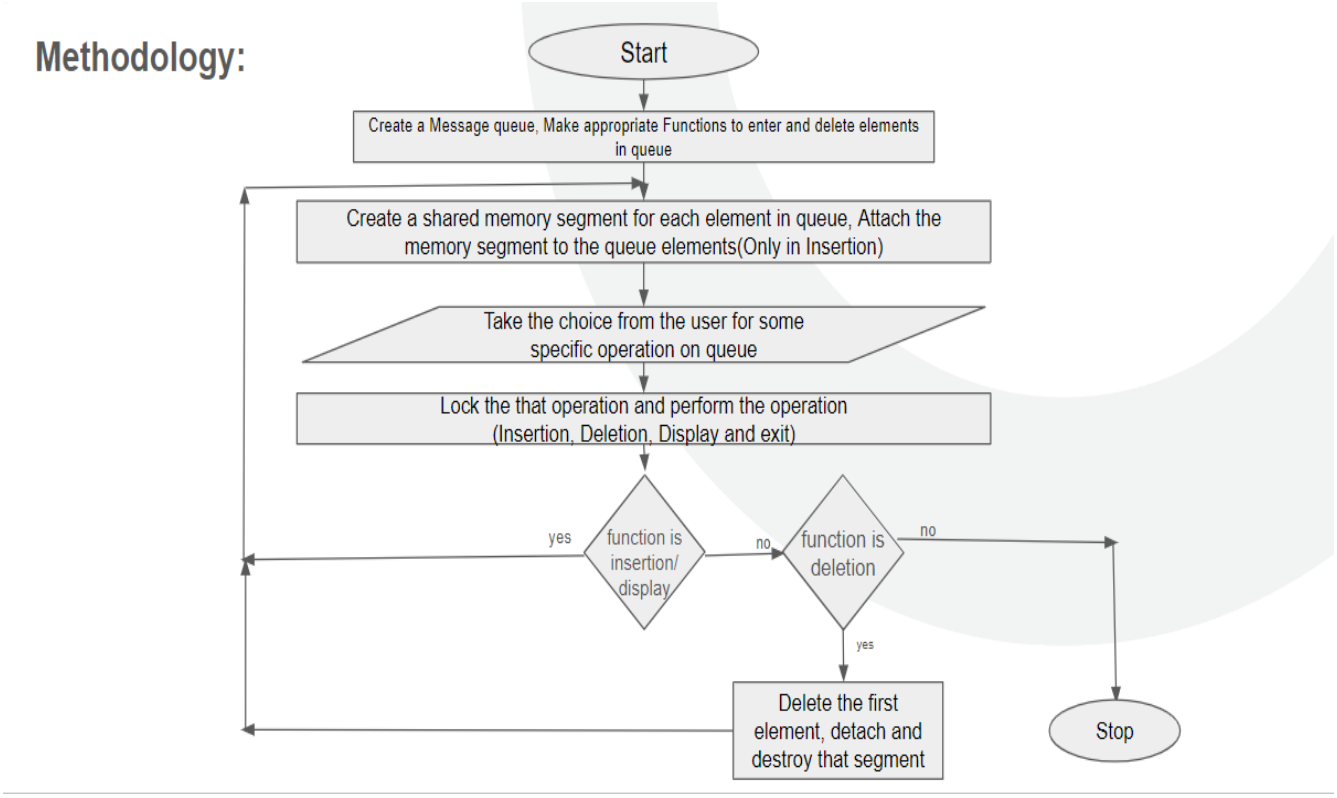
Windows Kernel: At the heart of the Windows operating system lies the Windows Kernel, which is part of the Windows NT architecture. The kernel provides core functionalities such as process and memory management, hardware abstraction, device drivers, and security features. It serves as the foundation upon which the entire operating system operates.

User Mode and Kernel Mode: Windows employs a dual-mode operation, where software executes either in user mode or kernel mode. User mode applications run with restricted access to system resources and rely on system services provided by the kernel through defined APIs. Kernel mode, on the other hand, allows direct access to hardware and privileged operations, primarily utilized by the operating system components and device drivers.

Win32 Subsystem: The Win32 subsystem is a key component of the Windows architecture, responsible for providing compatibility with 32-bit Windows applications. It includes the user-mode components necessary for running Win32-based applications, handling graphical user interface (GUI) interactions, and managing system resources..

Methodology

For Linux Based OS (UBUNTU)



For Windows :

1.Choose a Communication Protocol: Select a communication protocol for exchanging text messages between the two terminals. Common choices include TCP/IP, UDP, HTTP, WebSocket, or MQTT, depending on your specific requirements.
2.Set Up Terminal Devices: Ensure that both terminal devices (computers, smartphones, IoT devices, etc.) are properly configured for communication. They should have network connectivity (Wi-Fi, cellular, or Ethernet) and the necessary software and hardware components.
3.Develop or Choose Messaging Software: Decide whether you will develop custom messaging software or use existing messaging platforms and apps (e.g., WhatsApp, Signal, or Slack). Developing custom software gives you more control but requires more effort.
4.Implement Authentication and Security: Establish a secure authentication mechanism to verify the identity of the terminals. You may use passwords, API keys, or more advanced methods like OAuth or JWT for authentication.
5.Addressing and Routing: Assign unique identifiers or addresses to each terminal, which will be used to route messages. These addresses can be user IDs, device names, or IP addresses.
6.Message Encoding and Decoding: Define a standard message format (e.g., JSON, XML, or plain text) for encoding and decoding messages. Ensure that both terminals can understand and interpret this format.
7.Establish Connection: Use the selected communication protocol to establish a connection between the two terminals. For example, if using TCP/IP, one terminal can act as a server, and the other as a client. If using WebSocket, both can be clients or use a client-server model.
8.Send and Receive Messages: Develop the logic for sending and receiving text messages between the terminals. This includes message queuing, error handling, and message

acknowledgment to ensure reliability.
9.Implement Real-Time Updates: If real-time communication is required, implement mechanisms like long polling or WebSockets to enable instant message delivery without polling for updates continuously.
10.Logging and Monitoring: Implement logging and monitoring features to track message history, diagnose issues, and ensure system performance.

System Calls Used:

1. Shared Memory Initialization:
   - shm_open(): Opens or creates a shared memory object named "/message_queue".
   - ftruncate(): Sets the size of the shared memory object to the size of the struct queue.
   - mmap(): Maps the shared memory object into the process's address space.

2. Semaphore Initialization:
   - sem_open(): Opens or creates a named semaphore with the name "/mutex" for mutual exclusion.
   - sem_init(): Initializes the semaphore with the specified name and initial value (1).
   - sem_unlink(): Removes the named semaphore after it's no longer needed.

3. Forking:
   - fork(): Creates a child process. The child process handles user input and performs queue operations, while the parent process waits for the child to terminate.

4. Child Process (Producer/Consumer Logic):
   - In the child process (pid == 0):
     - Enters a loop to continuously prompt the user for choices.
     - Reads the user's choice and performs the corresponding operation (enqueue, dequeue, display, or exit).
     - Calls the respective functions (enqueue(), dequeue(), display()), passing the shared queue structure mq.

5. Parent Process (Wait for Child):
   - In the parent process (pid > 0), waits for the child process to terminate using wait(NULL).

6. Cleanup and Exit:
   - Upon termination, both parent and child processes perform cleanup:
     - Close semaphores using sem_close().
     - Unmap shared memory using munmap().
     - Unlink shared memory object and named semaphores using shm_unlink() and sem_unlink() respectively.

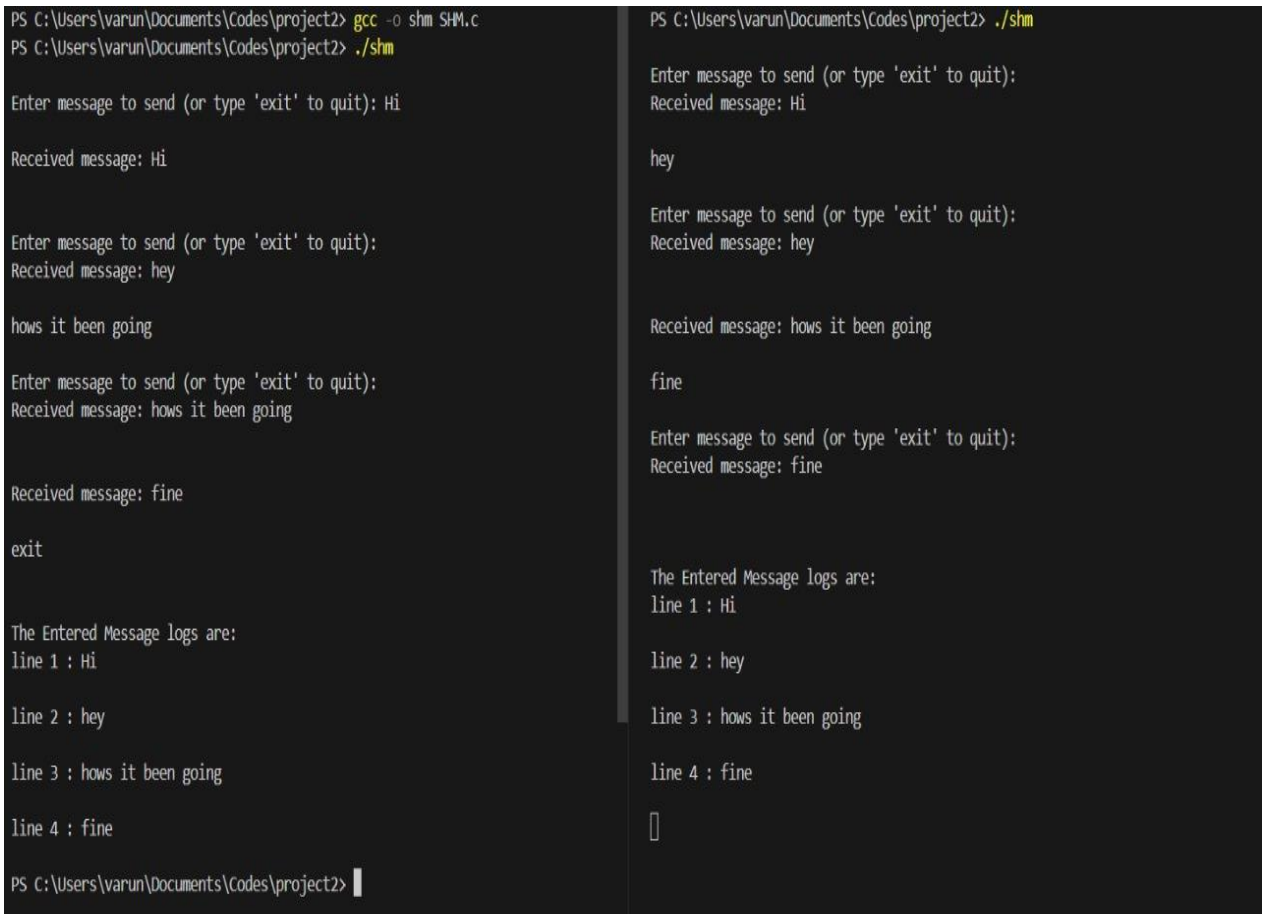
Result & Output:

Ubuntu:

```
varun@varun-VirtualBox:~/Desktop/codes$ ./f
MESSAGE QUEUE


1:Insert
2:Delete
3:Display
4:exit
Enter the Choice: 1
Enter the Message to be sent: Info_needed_asap

1:Insert
2:Delete
3:Display
4:exit
Enter the Choice: 1
Enter the Message to be sent: send_it

1:Insert
2:Delete
3:Display
4:exit
Enter the Choice: 3
Messages in Queue are:
required_info

1:Insert
2:Delete
3:Display
4:exit
Enter the Choice: 4
varun@varun-VirtualBox:~/Desktop/codes$
```

```
varun@varun-VirtualBox:~/Desktop/codes$ ./f
MESSAGE QUEUE


1:Insert
2:Delete
3:Display
4:exit
Enter the Choice: 3
Messages in Queue are:
Info_needed_asap
send_it

1:Insert
2:Delete
3:Display
4:exit
Enter the Choice: 1
Enter the Message to be sent: required_info

1:Insert
2:Delete
3:Display
4:exit
Enter the Choice: 2
Removed message is Info_needed_asap

1:Insert
2:Delete
3:Display
4:exit
Enter the Choice: 2
Removed message is send_it

1:Insert
2:Delete
3:Display
4:exit
Enter the Choice: 4
varun@varun-VirtualBox:~/Desktop/codes$
```

Windows:


PS C:\Users\varun\Documents\Codes\project2> gcc -o shm SHM.c
PS C:\Users\varun\Documents\Codes\project2> ./shm
Enter message to send (or type 'exit' to quit): Hi
Received message: Hi
Enter message to send (or type 'exit' to quit):
Received message: hey
hows it been going
Enter message to send (or type 'exit' to quit):
Received message: hows it been going
Received message: fine
exit
The Entered Message logs are:
line 1 : Hi
line 2 : hey
line 3 : hows it been going
line 4 : fine
PS C:\Users\varun\Documents\Codes\project2>
PS C:\Users\varun\Documents\Codes\project2> ./shm
Enter message to send (or type 'exit' to quit):
Received message: Hi
hey
Enter message to send (or type 'exit' to quit):
Received message: hey
Received message: hows it been going
fine
Enter message to send (or type 'exit' to quit):
Received message: fine
The Entered Message logs are:
line 1 : Hi
line 2 : hey
line 3 : hows it been going
line 4 : fine

Conclusion:

Throughout this journey, we will delve into the foundational concepts underpinning shared memory message queues, emphasizing their significance in modern computing paradigms. We will navigate through the principles of inter-process communication, explore the mechanics of shared memory management, and dissect the intricacies of system call invocation. Ultimately, our destination is the design and implementation of a robust shared memory message queue system, offering developers newfound avenues for optimizing communication between processes.

Applications:

1.Parallel Computing: In parallel computing environments, multiple processes often need to exchange data efficiently. Shared memory message queues facilitate communication among parallel processes, improving coordination and synchronization.

2.Networking: Shared memory message queues can be utilized in networking applications where multiple processes need to exchange data packets or messages. For instance, in network servers handling multiple client connections, message queues can improve throughput and responsiveness.

3.Operating Systems: Shared memory message queues are fundamental in operating systems for communication between system components or kernel modules. They facilitate interaction between user-space processes and the kernel, aiding in system management, resource allocation, and device control.

4.Real-time Systems: Real-time applications require precise timing and efficient communication mechanisms. Shared memory message queues provide a low-latency IPC mechanism suitable for real-time systems, such as industrial automation, robotics, and avionics.

5.Distributed Systems: In distributed computing environments, where processes run on different machines, shared memory message queues can be used in conjunction with network protocols for inter-node communication. They can enhance the performance and reliability of distributed applications by reducing network overhead and latency.

6.Multi-threaded Applications: Shared memory message queues can facilitate communication among threads within a process. In multi-threaded applications, they can be used to pass data or synchronize operations between threads efficiently.