Implementing a Shared Resource Manager using Semaphores

Description:

In this scenario, you are responsible for developing a messaging application that supports multiple users to communicate with each other. The objective is to ensure that the messages are delivered and displayed in the correct order, maintaining the chronological order of the conversation. To achieve this, you need to implement a shared resource manager using semaphores in C++.

The Messaging Application:

- The messaging application allows multiple users to interact with each other by sending messages.
- Each user has a unique identifier (User ID) to distinguish between them. • Users can send messages to other users, and each message includes the sender's ID and the content of the message.
- The messages need to be processed and displayed in the order they were sent to maintain the correct chronological order of the conversation.

The Shared Resource: Message Queue

- To manage message delivery, you will implement a shared resource called the "Message Queue."
- The Message Queue is a data structure that holds all messages sent by different users, and it follows the First-In-First-Out (FIFO) order.
- When a user sends a message, it is enqueued into the Message Queue. • The application will have multiple threads, each representing a different user in the system. • These user threads can simultaneously send messages to the Message Queue, potentially leading to race conditions.

Problem Statement:

- Your main objective is to ensure that messages are processed and displayed in the same order they were sent.
- To prevent race conditions and data inconsistency, you need to use semaphores to synchronize access to the Message Queue.
- The shared resource manager should provide the following operations: a. `enqueueMessage(int senderId, std::str message)` : This function adds a message to the end of the Message Queue with the sender's ID and the message content. b. `dequeueMessag e()` : This function removes and returns the front message from the Message Queue. c. `isEmpty` : This function returns true if the Message Queue is empty; otherwise,

it returns false.

Requirements:

• The shared resource manager using semaphores must ensure that multiple user threads can safely enqueue messages to the Message Queue without data corruption or race conditions. • The dequeue operation should correctly remove messages from the Message Queue in the same order they were added, maintaining the chronological order of the conversation. • Access to the Message Queue should be synchronized, and only one user thread should be allowed to enqueue or dequeue a message at a time to avoid conflicts and data inconsistency.

Implementation:

- You should design and implement the shared resource manager using semaphores to satisfy the requirements mentioned above.
- The solution should demonstrate correct usage of semaphores to ensure safe and synchronized access to the shared Message Queue.
- It is crucial to handle edge cases, such as checking if the Message Queue is empty before attempting to dequeue a message to avoid errors.
- The implementation should be efficient and minimize delays to ensure smooth communication between users.

By successfully implementing the shared resource manager using semaphores, you will enable users to send and receive messages in the correct order, creating a seamless messaging experience while maintaining data integrity and consistency.

Expected output

```
User A sends: "Hello from User A!"
User B sends: "Hi, this is User B."
User A sends: "Nice to meet you, User B!"
User C sends: "Hey, it's User C here."
User C sends: "User A, I got your message."
User B sends: "Hello, User C! Nice to meet you too."
User A sends: "User C, I got your message as well."
User C sends: "Great! Let's all chat together."
User B sends: "Sure, I'm up for it."
User C sends: "Me too."

Messages in the conversation:
1. User A: "Hello from User A!"
2. User B: "Hi, this is User B."
3. User A: "Nice to meet you, User B!"
4. User C: "Hey, it's User C here."
5. User C: "User A, I got your message."
6. User B: "Hello, User C! Nice to meet you too."
7. User A: "User C, I got your message as well."
8. User C: "Great! Let's all chat together."
9. User B: "Sure, I'm up for it."
10. User C: "Me too."
```