AVIJIT PAUL
ID: 00-30131

1. Is there any possibility of a deadlock occurring in this code? If so, under what conditions could it happen, and what could be potential solutions to avoid it?

Answer: A deadlock is a situation in which two or more threads in a multithreaded program are blocked, each waiting for a resource that the other thread holds. As a result, none of the threads can make progress, and the program effectively halts, unable to proceed further.

So I think, No, the deadlock may not arise as the synchronization mechanisms and the proper usage of notifyAll() calls appear to prevent the threads from getting permanently blocked. The design of the code seems to handle the producer-consumer scenario correctly. As both threads run sequentially and each time, after producing products, and consuming products -> it notifies all other threads. And the buffer is locked during the processing operation of any particular thread so that no other thread can manipulate it's data. However, some irregularities may show if we run the production loop way more than the current condition.

2. The producer code limits the buffer size to maxSize, but what happens if the producer thread attempts to add an item when the buffer is already full? How is this situation handled?

Answer: If the producer thread attempts to add an item when the buffer is already full, the producer thread enters the synchronization block where it locks the buffer object. Then the thread checks if the buffer is full. If it is full then the thread enters a waiting state using "buffer.wait()". This releases the lock on buffer and allow other threads to access. While in the waiting state, the producer thread stays inactive until it is get activated by "buffer.notifyAll() from consumer. Then the producer thread gets activated and cheks the condition of the loop again. If the buffer is full at that time then it goes again to the waiting state using buffer.wait(). But if the buffer is no longer full then the producer thread exits the loop and adds the item to the buffer. It then calls "buffer.notifyAll()' to wake up any other waiting threads that were blocked by empty buffer.
This gives the surety that the producer will not add an item to the buffer if it is already full. Instead it waits for the consumer to consume items.

3. The consumer code removes items from the buffer, but what happens if the consumer thread attempts to remove an item when the buffer is empty? How is this situation handled?

Answer: If the consumer thread attempts to remove an item from the buffer when it's empty then

The consumer thread enters a waiting state using buffer.wait(). It remains inactive until the producer thread adds items to the buffer and calls buffer.notifyAll() to wake up waiting threads. Once awakened, the consumer thread checks if the buffer is still empty, and if so, it goes back to waiting. When the buffer has items, the consumer thread proceeds to remove and process an item, and then calls buffer.notifyAll() to potentially wake up waiting threads.
This synchronization mechanism ensures the consumer thread only consumes items when they are available in the buffer and prevents it from trying to consume from an empty buffer.

4. Both the producer and consumer use synchronization on the buffer object to ensure thread safety. What purpose does this synchronization serve, and why is it necessary?

Answer: Both the producer and consumer use synchronization on the buffer object to ensure thread safety. This synchronization serves the purpose of preventing concurrent access to shared resources, such as the buffer in this case. It's necessary to avoid data corruption, race conditions, and ensure consistent behavior in a multi-threaded environment.

5. The code uses buffer.notifyAll() to wake up waiting threads. Why is notifyAll() used instead of notify()? What could be the potential issues if notify() were used instead?

Answer: The code uses buffer.notifyAll() to wake up waiting threads instead of notify(). This is because notifyAll() wakes up all waiting threads, ensuring that no threads are inadvertently left waiting. If notify() were used instead, there could be potential issues such as some waiting threads not being awakened, which might lead to thread starvation or unpredictable behavior in the program.