Receive Buffer Size: 8192

Client connected: /192.168.1.100

Send Buffer Size: 8192

Receive Buffer Size: 8192

Lab 4: Incrementing a counter in shared memory.

Explanation:

Incrementing a counter in shared memory refers to the process of increasing the value of a counter variable that is stored in a region of memory that is accessible to multiple threads or processes concurrently. Shared memory is a form of inter-process communication (IPC) that allows different processes or threads to access and manipulate the same memory region, making it a useful mechanism for synchronization and coordination in concurrent programming.

Here's a general overview of how incrementing a counter in shared memory might work:

Initialization: Initially, you would create or allocate a block of shared memory that can hold the counter variable. This shared memory block is typically created by one process or thread and then shared among multiple processes or threads.

Access Control: To ensure that multiple processes or threads don't try to access and modify the shared counter simultaneously (which could lead to race conditions and data corruption), synchronization mechanisms like mutexes, semaphores, or atomic operations are often used. These mechanisms provide a way to control access to the shared memory region.

Increment Operation: When a process or thread wants to increment the counter, it first acquires the necessary synchronization mechanism to gain exclusive access to the shared memory. Once it has exclusive access, it can safely read the current value of the counter, increment it, and write the updated value back to the shared memory location.

Release Lock: After incrementing the counter and updating the shared memory, the process or thread releases the synchronization mechanism, allowing other processes or threads to access the counter if they need to.

Program Code:

In Java, you can use the java.util.concurrent package to implement a counter increment operation in shared memory. The package provides thread-safe constructs for synchronization, including AtomicInteger for atomic operations on integers and Lock for more fine-grained control.

```
import java.util.concurrent.atomic.AtomicInteger;

public class SharedMemoryCounter {
    public static void main(String[] args) {
        // Create an AtomicInteger with an initial value of 0
        AtomicInteger counter = new AtomicInteger(0);

        // Create multiple threads to increment the counter
        int numThreads = 4;
        Thread[] threads = new Thread[numThreads];

        for (int i = 0; i < numThreads; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 10000; j++) {
                    // Increment the counter atomically
                    counter.incrementAndGet();
                }
            });
            threads[i].start();
        }

        // Wait for all threads to finish
        try {
            for (Thread thread : threads) {
                thread.join();
```

```
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Print the final value of the counter
        System.out.println("Final Counter Value: " + counter.get());
    }
}
```

In the above program we create an AtomicInteger called counter and use the incrementAndGet() method to increment it atomically from multiple threads. The example creates four threads, each incrementing the counter 10,000 times. Finally, we wait for all threads to finish and print the final value of the counter. The use of AtomicInteger ensures that the counter is incremented safely in a multi-threaded environment without the need for explicit synchronization mechanisms like locks.

Lab 5: