1. SOLUTION OVERVIEW

    - Uses shared memory for data storage
    - Uses semaphores for synchronization
    - Implements circular buffer pattern
    - Uses separate processes for producer and consumer

2. COMPONENTS

    A. Shared Memory Structure:

    ```
    struct shared_data {
        int producer_index;  // Write position
        int consumer_index;  // Read position
        int buffer[10];      // Data storage
    }
    ```

    B. Semaphores (4 total):

    ```
    semid[0] = 10  // Empty slots counter
    semid[1] = 0   // Full slots counter
    semid[2] = 1   // Producer mutex
    semid[3] = 1   // Consumer mutex
    ```

3. IMPLEMENTATION FLOW

    A. Initialization (create.c):

    - Creates shared memory segment
    - Initializes indices to 0
    - Creates and initializes semaphores
    - Sets up initial environment

    B. Producer Process (producer.c):

    - Waits for empty slot (P(empty))
    - Gets exclusive access (P(prod_mutex))
    - Writes data to buffer
    - Updates producer_index

  - Releases mutex (V(prod_mutex))
  - Signals slot filled (V(full))

  C. Consumer Process (consumer.c):

  - Waits for full slot (P(full))
  - Gets exclusive access (P(cons_mutex))
  - Reads data from buffer
  - Updates consumer_index
  - Releases mutex (V(cons_mutex))
  - Signals slot emptied (V(empty))

4. SYNCHRONIZATION MECHANISM

   A. Empty/Full Management:

   - Empty semaphore (semid[0]) prevents buffer overflow
   - Full semaphore (semid[1]) prevents reading empty buffer

   B. Mutual Exclusion:

   - Producer mutex (semid[2]) protects producer operations
   - Consumer mutex (semid[3]) protects consumer operations

5. SEQUENCE OF OPERATIONS

   Producer:

   ```
   P(empty)         // Wait for space
   P(prod_mutex)    // Enter critical section
   write_data()     // Modify buffer
   V(prod_mutex)    // Exit critical section
   V(full)          // Signal data available
   ```

   Consumer:

```
P(full)          // Wait for data
P(cons_mutex)    // Enter critical section
read_data()      // Access buffer
V(cons_mutex)    // Exit critical section
V(empty)         // Signal space available
```

6. KEY FEATURES

- Circular buffer implementation
- Safe concurrent access
- No buffer overflow/underflow
- Proper resource management
- Automatic cleanup with SEM_UNDO

7. USAGE

```
# Compile
gcc create.c -o create
gcc producer.c -o producer
gcc consumer.c -o consumer

# Execute (in separate terminals)
./create
./producer
./consumer
```

8. IMPORTANT CONSIDERATIONS

- Create process must run first
- Semaphore operations must maintain proper order
- Buffer indices must wrap around correctly
- Shared data access must be protected
- Resource limits must be respected