Producer-Consumer Implementation Documentation Overview

This documentation describes a classic producer-consumer implementation using shared memory and semaphores in C. The solution consists of two separate programs:

1. **Producer**: Generates random items and places them in a shared circular buffer
2. **Consumer**: Retrieves items from the shared buffer and processes them

The implementation uses POSIX shared memory and semaphores to handle synchronization between the two processes.

System Architecture

Shared Resources

- **Shared Memory**: A circular buffer with capacity for 2 items
- **Semaphores**:
    - mutex: Ensures mutual exclusion when accessing the shared buffer
    - empty: Counts available slots in the buffer (initialized to buffer size)
    - full: Counts items in the buffer (initialized to 0)

Data Structure

```
typedef struct {
   int buffer[TABLE_SIZE];  // Fixed-size buffer (size = 2)
   int in;    // Next position to insert
   int out;   // Next position to remove
   int count; // Current number of items
} SharedTable;
```

Producer Implementation

The producer program:

1. Creates and initializes the shared memory segment
2. Creates and initializes the required semaphores
3. Generates 10 random integer items (0-99)
4. Places items in the shared buffer
5. Cleans up resources when done

Key Functions

- producer_thread(): Main function that generates random items and places them in the buffer
- main(): Sets up shared resources and manages the producer thread

Producer Algorithm

1. Wait if buffer is full (decrement 'empty' semaphore)

2. Acquire mutex to access shared memory

3. Generate a random item and place it in the buffer at position 'in'

4. Update 'in' pointer using modulo arithmetic for circular buffer

5. Release mutex

6. Signal that a new item is available (increment 'full' semaphore)

7. Wait briefly before producing next item

Consumer Implementation

The consumer program:

1. Opens the existing shared memory segment
2. Opens the existing semaphores
3. Consumes 10 items from the shared buffer
4. Closes (but does not unlink) shared resources when done

Key Functions

- consumer_thread(): Main function that retrieves items from the buffer
- main(): Connects to shared resources and manages the consumer thread

Consumer Algorithm

1. Wait if buffer is empty (decrement 'full' semaphore)

2. Acquire mutex to access shared memory

3. Retrieve item from buffer at position 'out'

4. Update 'out' pointer using modulo arithmetic for circular buffer

5. Release mutex

6. Signal that a buffer slot is now available (increment 'empty' semaphore)

7. Wait briefly before consuming next item

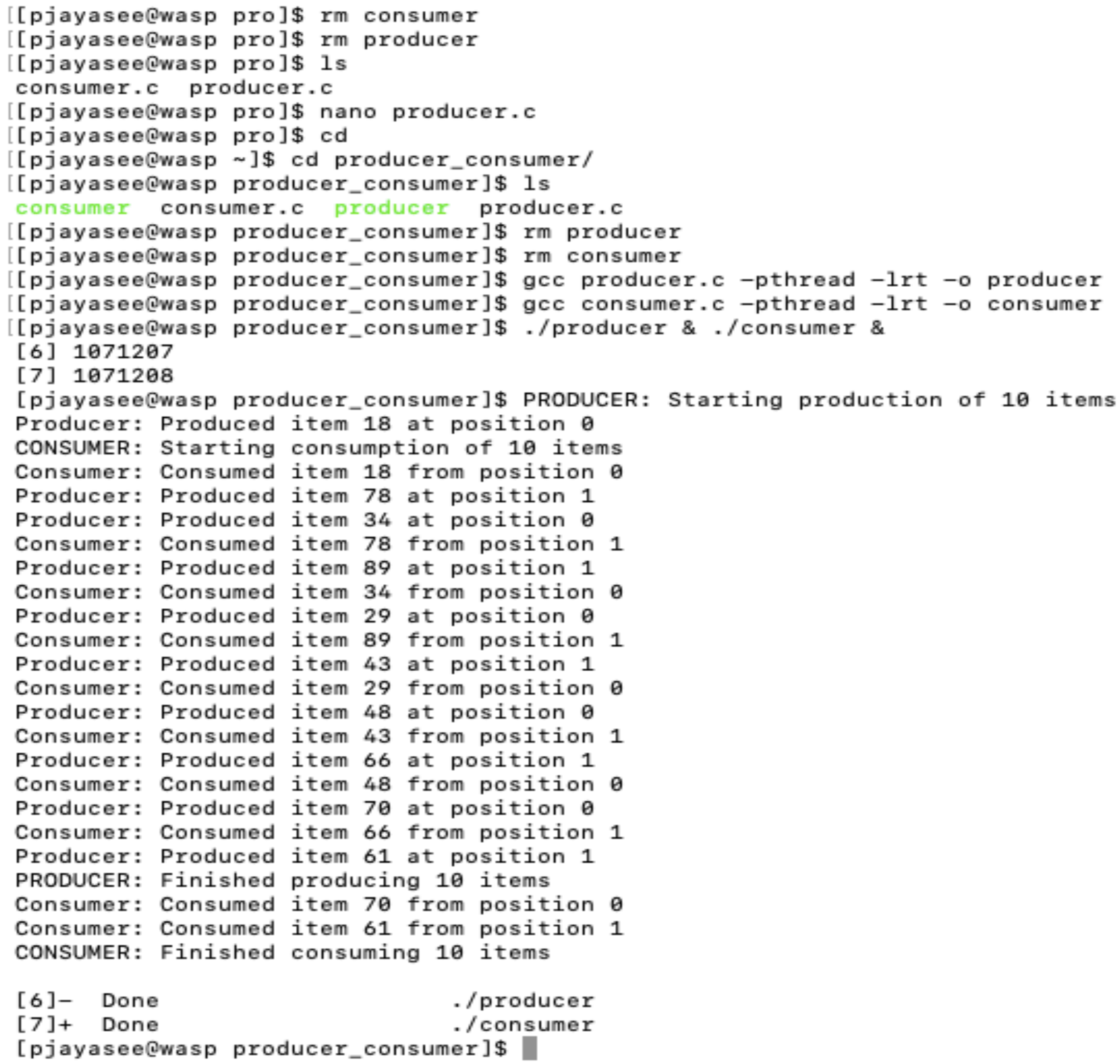
Synchronization Mechanism

The implementation uses semaphores to handle three critical aspects:

1. **Mutual Exclusion**: The mutex semaphore ensures that only one process can access the shared buffer at any time.
2. **Buffer Full Condition**: The empty semaphore prevents the producer from adding items when the buffer is full.
3. **Buffer Empty Condition**: The full semaphore prevents the consumer from retrieving items when the buffer is empty.

SAMPLE OUTPUT

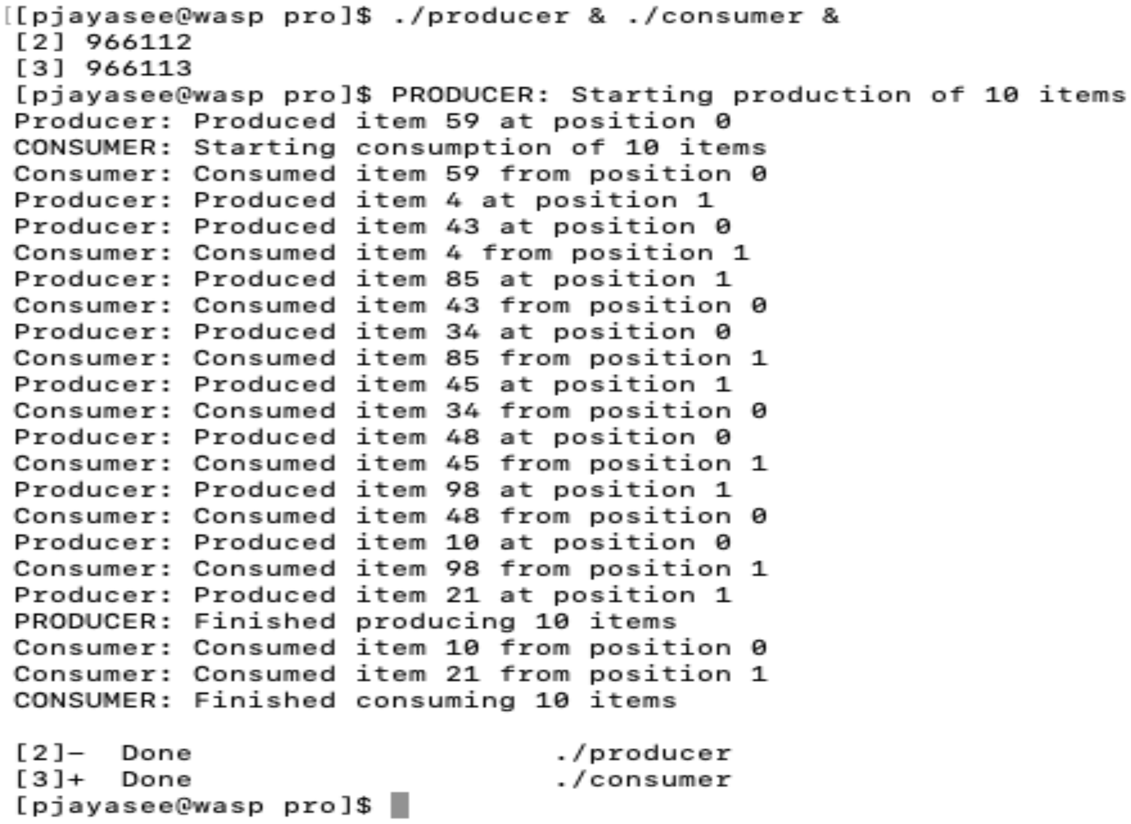
ATTEMPT 1 AND SUCCESS



```
[[pjayasee@wasp pro]$ rm consumer
[[pjayasee@wasp pro]$ rm producer
[[pjayasee@wasp pro]$ ls
 consumer.c  producer.c
[[pjayasee@wasp pro]$ nano producer.c
[[pjayasee@wasp pro]$ cd
[[pjayasee@wasp ~]$ cd producer_consumer/
[[pjayasee@wasp producer_consumer]$ ls
 consumer  consumer.c  producer  producer.c
[[pjayasee@wasp producer_consumer]$ rm producer
[[pjayasee@wasp producer_consumer]$ rm consumer
[[pjayasee@wasp producer_consumer]$ gcc producer.c -pthread -lrt -o producer
[[pjayasee@wasp producer_consumer]$ gcc consumer.c -pthread -lrt -o consumer
[[pjayasee@wasp producer_consumer]$ ./producer & ./consumer &
 [6] 1071207
 [7] 1071208
 [pjayasee@wasp producer_consumer]$ PRODUCER: Starting production of 10 items
 Producer: Produced item 18 at position 0
 CONSUMER: Starting consumption of 10 items
 Consumer: Consumed item 18 from position 0
 Producer: Produced item 78 at position 1
 Producer: Produced item 34 at position 0
 Consumer: Consumed item 78 from position 1
 Producer: Produced item 89 at position 1
 Consumer: Consumed item 34 from position 0
 Producer: Produced item 29 at position 0
 Consumer: Consumed item 89 from position 1
 Producer: Produced item 43 at position 1
 Consumer: Consumed item 29 from position 0
 Producer: Produced item 48 at position 0
 Consumer: Consumed item 43 from position 1
 Producer: Produced item 66 at position 1
 Consumer: Consumed item 48 from position 0
 Producer: Produced item 70 at position 0
 Consumer: Consumed item 66 from position 1
 Producer: Produced item 61 at position 1
 PRODUCER: Finished producing 10 items
 Consumer: Consumed item 70 from position 0
 Consumer: Consumed item 61 from position 1
 CONSUMER: Finished consuming 10 items

 [6]-  Done                    ./producer
 [7]+  Done                    ./consumer
 [pjayasee@wasp producer_consumer]$ 
```

- **ATTEMPT 2 AND SUCCESS**



```
[[pjayasee@wasp pro]$ ./producer & ./consumer &
[2] 966112
[3] 966113
[pjayasee@wasp pro]$ PRODUCER: Starting production of 10 items
Producer: Produced item 59 at position 0
CONSUMER: Starting consumption of 10 items
Consumer: Consumed item 59 from position 0
Producer: Produced item 4 at position 1
Producer: Produced item 43 at position 0
Consumer: Consumed item 4 from position 1
Producer: Produced item 85 at position 1
Consumer: Consumed item 43 from position 0
Producer: Produced item 34 at position 0
Consumer: Consumed item 85 from position 1
Producer: Produced item 45 at position 1
Consumer: Consumed item 34 from position 0
Producer: Produced item 48 at position 0
Consumer: Consumed item 45 from position 1
Producer: Produced item 98 at position 1
Consumer: Consumed item 48 from position 0
Producer: Produced item 10 at position 0
Consumer: Consumed item 98 from position 1
Producer: Produced item 21 at position 1
PRODUCER: Finished producing 10 items
Consumer: Consumed item 10 from position 0
Consumer: Consumed item 21 from position 1
CONSUMER: Finished consuming 10 items

[2]-  Done                    ./producer
[3]+  Done                    ./consumer
[pjayasee@wasp pro]$ 
```

Circular Buffer Behavior

The implementation uses a circular buffer with positions 0 and 1:

- The producer places items at position in and increments in (with wraparound)
- The consumer retrieves items from position out and increments out (with wraparound)
- Wraparound is achieved using modulo arithmetic: position = (position + 1) % TABLE_SIZE

Resource Management

Both programs properly manage system resources:

- **Producer**:
  - Creates and initializes all shared resources
  - Cleans up all resources when done (unlinks shared memory and semaphores)
- **Consumer**:
  - Opens existing shared resources
  - Closes but does not unlink shared resources (since producer handles cleanup)

Compilation and Execution

To compile the programs:

```
gcc producer.c -o producer -pthread -lrt
gcc consumer.c -o consumer -pthread -lrt
```

To run the programs:

```
./producer & ./consumer &
```

Notes and Observations

- The buffer size is intentionally small (2) to demonstrate the synchronization mechanisms.
- The producer has a 1-second delay between items, while the consumer has a 2-second delay, creating different production/consumption rates to test the synchronization.
- The circular buffer implementation correctly handles wraparound when the buffer is filled and emptied multiple times.