✅ Theory: Producer–Consumer Problem

📌 What is the Producer–Consumer Problem?

It is a classic **process synchronization problem** where:

- **Producer** generates data and puts it into a **shared buffer**.
- **Consumer** removes data from the buffer and processes it.
- The buffer has **limited size**, so:
  - Producer must **wait** if the buffer is **full**.
  - Consumer must **wait** if the buffer is **empty**.

🔒 What is Synchronization Needed?

Because both producer and consumer **access shared memory (buffer)** at the same time → may cause race conditions.

🛠️ What Tools Are Used?

Concept	Used As	Purpose
Semaphore `emptySlots`	Counting semaphore	Tracks **spaces available** in buffer
Semaphore `fullSlots`	Counting semaphore	Tracks **items available** in buffer
Mutex `lock`	Binary lock	Ensures **mutual exclusion** while accessing buffer (prevents race condition)
Threads `pthread`	Producer & Consumer	Runs both processes in parallel

---

✅ Program Explanation (Line by Line)

➤ Header Files

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
```

- `stdio.h` → `printf()`
- `pthread.h` → thread creation
- `semaphore.h` → semaphores
- `unistd.h` → `sleep()`

➤ Buffer and Control Variables

```
#define SIZE 5 // Buffer size
int buffer[SIZE];
int in = 0, out = 0;
```

- `buffer[SIZE]` → Shared circular queue (size = 5)
- `in` → index where producer inserts
- `out` → index where consumer removes

➤ Semaphores and Mutex Declaration

```
sem_t emptySlots; // counts empty slots
sem_t fullSlots;  // counts full slots
pthread_mutex_t lock; // mutual exclusion
```

✅ Producer Function

```
void *producer(void *arg) {
    for (int i = 0; i < 10; i++) {
        int item = rand() % 100;   // Produce item

        sem_wait(&emptySlots);     // Wait if buffer is full
        pthread_mutex_lock(&lock); // Lock shared buffer

        buffer[in] = item;
        printf("Producer produced: %d\n", item);
        in = (in + 1) % SIZE;      // Move circularly

        pthread_mutex_unlock(&lock);
        sem_post(&fullSlots);      // Increase filled slot count
        sleep(1);                  // Simulate time delay
    }
    return NULL;
}
```

🔍 Breakdown:

✅ `rand()` → Generate random number (item)
✅ `sem_wait(emptySlots)` → Producer waits if no space in buffer
✅ `pthread_mutex_lock()` → Protect buffer from race condition
✅ Insert item → `buffer[in] = item;`
✅ Circular increment → prevents overflow

✅ `sem_post(fullSlots)` → Increase count of filled slots
✅ `sleep(1)` → Simulates production time

✅ Consumer Function

```
void *consumer(void *arg) {
    for (int i = 0; i < 10; i++) {

        sem_wait(&fullSlots);      // Wait if buffer is empty
        pthread_mutex_lock(&lock);

        int item = buffer[out];
        printf("Consumer consumed: %d\n", item);
        out = (out + 1) % SIZE;

        pthread_mutex_unlock(&lock);
        sem_post(&emptySlots);     // Increase empty slot count
        sleep(2);
    }
    return NULL;
}
```

🔍 Breakdown:

✅ `sem_wait(fullSlots)` → Wait until buffer has an item
✅ Extract item from buffer
✅ Print consumed value
✅ `sem_post(emptySlots)` → Space freed
✅ `sleep(2)` → Simulates slower consumer

✅ Main Function

```
int main() {
    pthread_t prodThread, consThread;

    sem_init(&emptySlots, 0, SIZE); // Initially all slots empty
    sem_init(&fullSlots, 0, 0);     // No data produced yet
    pthread_mutex_init(&lock, NULL);

    pthread_create(&prodThread, NULL, producer, NULL);
    pthread_create(&consThread, NULL, consumer, NULL);

    pthread_join(prodThread, NULL);
    pthread_join(consThread, NULL);

    sem_destroy(&emptySlots);
```

```
    sem_destroy(&fullSlots);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

🔍 Breakdown:

Function	Purpose
`sem_init(&emptySlots, 0, SIZE)`	Buffer empty → SIZE empty slots available
`sem_init(&fullSlots, 0, 0)`	No data produced yet
`pthread_create()`	Start producer and consumer threads
`pthread_join()`	Wait until threads finish
`sem_destroy()`	Cleanup memory
`pthread_mutex_destroy()`	Destroy lock