K.GOWTHAM-192372078

18. Construct a C program to simulate producer-consumer problem using semaphores.

AIM:

To construct a C program to simulate the Producer-Consumer problem using semaphores, ensuring synchronization between the producer and consumer processes while preventing race conditions and buffer overflows or underflows.

ALGORITHM:

1. **Initialization:**
   - Define a shared buffer with a fixed size.
   - Initialize three semaphores:
     - empty: Counts the number of available slots in the buffer (initially equal to the buffer size).
     - full: Counts the number of filled slots in the buffer (initially zero).
     - mutex: Ensures mutual exclusion for buffer access (initialized to 1).
2. **Producer Process:**
   - Repeatedly execute the following steps:

1. Wait (sem_wait) on the empty semaphore to ensure a free slot is available.
2. Wait (sem_wait) on the mutex semaphore to gain exclusive access to the buffer.
3. Produce an item and place it in the buffer.
4. Signal (sem_post) the mutex semaphore to release the buffer.
5. Signal (sem_post) the full semaphore to indicate a filled slot.

3. **Consumer Process:**
   - Repeatedly execute the following steps:

1. Wait (sem_wait) on the full semaphore to ensure a filled slot is available.
2. Wait (sem_wait) on the mutex semaphore to gain exclusive access to the buffer.
3. Remove an item from the buffer for consumption.
4. Signal (sem_post) the mutex semaphore to release the buffer.
5. Signal (sem_post) the empty semaphore to indicate a free slot.

4. **Concurrent Execution:**
   - Create separate threads for the producer and consumer processes.
   - Ensure both threads run concurrently and modify the shared buffer as per their respective logic.
5. **Termination:**
   - Stop the producer and consumer threads after a predefined number of operations or based on user input.
   - Destroy all semaphores to release system resources.

PROCEDURE:

1. **Start:**
   Initialize necessary variables, shared buffer, and semaphores.
2. **Define Semaphores:**
   - Create a semaphore empty initialized to the buffer size to track available slots.
   - Create a semaphore full initialized to 0 to track filled slots.

   - Create a semaphore mutex initialized to 1 to enforce mutual exclusion.

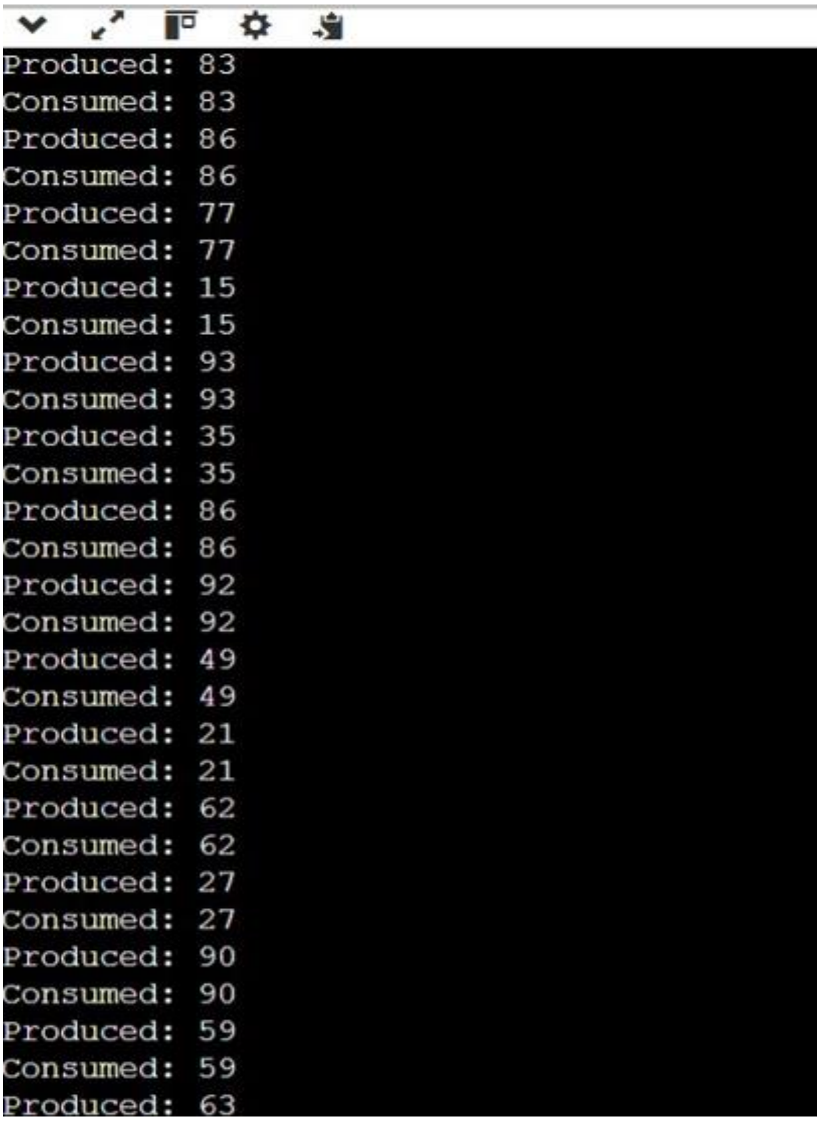
3. **Define Shared Buffer:**
   - Set up a circular buffer with a fixed size.
   - Use in and out pointers to manage the producer and consumer operations.
4. **Create Producer Thread:**
   - In the producer thread:
     - Wait on empty and mutex semaphores.
     - Produce an item and insert it into the buffer at the in index.
     - Update the in index to the next position in a circular manner.
     - Signal the mutex and full semaphores to indicate a successful operation.
5. **Create Consumer Thread:**
   - In the consumer thread:
     - Wait on full and mutex semaphores.
     - Consume an item from the buffer at the out index.
     - Update the out index to the next position in a circular manner.
     - Signal the mutex and empty semaphores to indicate a successful operation.
6. **Run Threads Concurrently:**
   - Execute both producer and consumer threads concurrently using pthread_create.
7. **Synchronization:**
   - Ensure that both threads operate in sync by using semaphores to handle mutual exclusion and resource tracking.
8. **Stop and Cleanup:**
   - Terminate the threads after a fixed number of operations.
   - Destroy the semaphores to release resources.
9. **End:**
   Stop the program after all operations are completed.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#ifdef _WIN32
#include <windows.h> // For Sleep on Windows
#else
#include <unistd.h> // For sleep on Unix-like systems
#endif

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0;

sem_t empty, full, mutex;
```

```
void *producer(void *param) {
    int item;
    while (1) {
        item = rand() % 100;
        sem_wait(&empty);
        sem_wait(&mutex);

        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;

        printf("Produced: %d\n", item);

        sem_post(&mutex);
        sem_post(&full);

        #ifdef _WIN32
        Sleep(1000); // Sleep for 1 second on Windows
        #else
        sleep(1); // Sleep for 1 second on Unix-like systems
        #endif
    }
}

void *consumer(void *param) {
    int item;
    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);

        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        printf("Consumed: %d\n", item);

        sem_post(&mutex);
        sem_post(&empty);

        #ifdef _WIN32
        Sleep(1000); // Sleep for 1 second on Windows
        #else
        sleep(1); // Sleep for 1 second on Unix-like systems
        #endif
    }
}

int main() {
    pthread_t prod, cons;

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

OUTPUT:



```
Produced: 83
Consumed: 83
Produced: 86
Consumed: 86
Produced: 77
Consumed: 77
Produced: 15
Consumed: 15
Produced: 93
Consumed: 93
Produced: 35
Consumed: 35
Produced: 86
Consumed: 86
Produced: 92
Consumed: 92
Produced: 49
Consumed: 49
Produced: 21
Consumed: 21
Produced: 62
Consumed: 62
Produced: 27
Consumed: 27
Produced: 90
Consumed: 90
Produced: 59
Consumed: 59
Produced: 63
```