Name: Khan Faisal
Roll no :- 197
Batch :- C12

SAP ID :- 60004240014
Subj :- OS

EXPERIMENT. (5)

* Aim :- Implement program on Producer & consumer with concurrent control.

* Theory :-

- Concurrency control ensures that multiple processes or threads execute correctly & efficiently when accessing shared resources without proper control, issues like race condition, deadlock & inconsistently data may arise.

- Key concepts :-

1. Race condition :-
Occurs when two or more process access shared data simultaneously, leading to unpredictable results
Ex :- Two threads incrementing the same count may miss some increment.

2. ~~Race condition~~ critical section :-
A code segment where shared resources are accessed. Must be protected to allow any one process at a time.

3. Solutions :-
1] Mutex Locks :- Ensures mutual exclusion (only one threads enters the critical section.)

FOR EDUCATIONAL USE

2] Semaphores :- Generalization of mutex with counting
ex:- limiting access to N.

3] Monitors : High level synchronization construct (lik in java)

• Semaphores :-

Binary Semaphore (0 to 1)

Counting Semaphore (-∞ to ∞)

Binary Semaphore are used to check for critical section is occupied or not whereas counting semaphores are used for counting full or empty slots of critical region.

• Simple analogy :-

Imagine a shared printer (CS) in an office

i] Only one employee can print at a time

ii] If multiple try to print they must wait in a Queue

• Conclusion :-

Thus, we performed and implemented program on semaphore of concurrent execution using threads using both Binary & counting semaphores.

FOR EDUCATIONAL USE

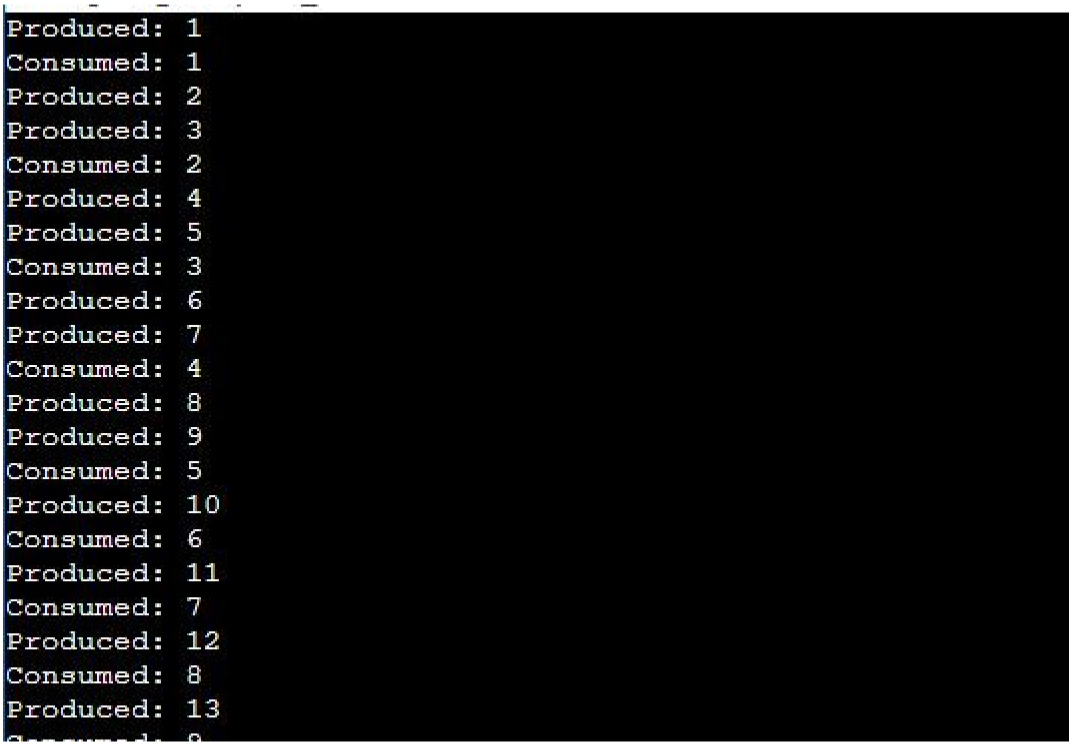
Program (Concurrent Execution):

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

#define BUFFER_SIZE 5
#define MAX_ITEMS 20  // Total items to produce

int buffer[BUFFER_SIZE];
int count = 0;
int in = 0;  // Producer index
int out = 0; // Consumer index

sem_t empty, full;
pthread_mutex_t mutex;

void* producer(void* arg) {
   for (int i = 1; i <= MAX_ITEMS; i++) {
      sem_wait(&empty);  // Wait for empty slot
      pthread_mutex_lock(&mutex);

      // Produce item
      buffer[in] = i;
      in = (in + 1) % BUFFER_SIZE;
      count++;
      printf("Produced: %d\n", i);

      pthread_mutex_unlock(&mutex);
      sem_post(&full);  // Signal new item available
      sleep(1);  // Simulate production time
   }
   return NULL;
}
```

```
void* consumer(void* arg) {
    for (int i = 1; i <= MAX_ITEMS; i++) {
        sem_wait(&full);  // Wait for available item
        pthread_mutex_lock(&mutex);

        // Consume item
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        printf("Consumed: %d\n", item);

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);  // Signal empty slot available
        sleep(2);    // Simulate consumption time (slower than
production)
    }
    return NULL;
}

int main() {
    pthread_t prod_thread, cons_thread;

    // Initialize semaphores and mutex
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    // Create threads
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);
```

```
    // Cleanup
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    printf("Finished producing and consuming %d items\n", MAX_ITEMS);
    return 0;
}
```

Output:



```
Produced: 1
Consumed: 1
Produced: 2
Produced: 3
Consumed: 2
Produced: 4
Produced: 5
Consumed: 3
Produced: 6
Produced: 7
Consumed: 4
Produced: 8
Produced: 9
Consumed: 5
Produced: 10
Consumed: 6
Produced: 11
Consumed: 7
Produced: 12
Consumed: 8
Produced: 13
```