Course Name: Operating systems
LAB: 04
Submitted By: Ebaad Khan
Roll: DT-22045

1) Implement the above code and paste the screen shot of the output.

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

int main() {
   int buffer[10], bufsize, in, out, produce,
consume, choice = 0;
   sem_t mutex, empty, full;

   in = 0;
   out = 0;
   bufsize = 10;

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

   while(choice != 3) {
      printf("\n1. Produce \t2. Consume \t3.
Exit");
      printf("\nEnter your choice: ");
      scanf("%d", &choice);

      switch(choice) {
         case 1:
            if (sem_trywait(&empty) == 0) {
               sem_wait(&mutex);
               printf("\nEnter the value: ");
               scanf("%d", &produce);
               buffer[in] = produce;
               in = (in + 1) % bufsize;
               sem_post(&mutex);
               sem_post(&full);
            } else {
               printf("\nBuffer is Full");
            }
            break;
case 2:
if (sem_trywait(&full) == 0) {
sem_wait(&mutex);
consume = buffer[out];
printf("\nThe consumed value is %d", consume);
out = (out + 1) % bufsize;
sem_post(&mutex);
sem_post(&empty);
} else {
printf("\nBuffer is Empty");
}
break;
}
}

sem_destroy(&mutex);
sem_destroy(&empty);
sem_destroy(&full);
return 0;
}
```

Output

```
1. Produce  2. Consume  3. Exit
Enter your choice: 1

Enter the value: 2

1. Produce  2. Consume  3. Exit
Enter your choice: 4

1. Produce  2. Consume  3. Exit
Enter your choice: 2

The consumed value is 2
1. Produce  2. Consume  3. Exit
Enter your choice: 2

Buffer is Empty
1. Produce  2. Consume  3. Exit
Enter your choice: 3


=== Code Execution Successful ===
```

2) Solve the producer-consumer problem using linked list. (You can perform this task using any
programming language)
Note: Keep the buffer size to 10 places.

```
import threading
import time
import random
from collections import deque

BUFFER_SIZE = 10

# Shared resources
buffer = deque(maxlen=BUFFER_SIZE)
mutex = threading.Semaphore(1)
empty = threading.Semaphore(BUFFER_SIZE)
full = threading.Semaphore(0)

class Producer(threading.Thread):
    def run(self):
        while True:
            item = random.randint(1, 100)
            empty.acquire()
            mutex.acquire()

            buffer.append(item)
            print(f"Produced {item} | Buffer:
{list(buffer)}")

            mutex.release()
            full.release()
            time.sleep(random.random())
```

OUTPUT

```
Produced 21 | Buffer: [21]
Consumed 21 | Buffer: []
Produced 62 | Buffer: [62]
Produced 3 | Buffer: [62, 3]
Consumed 62 | Buffer: [3]
Produced 84 | Buffer: [3, 84]
Consumed 3 | Buffer: [84]
Produced 42 | Buffer: [84, 42]
```

```
class Consumer(threading.Thread):
 def run(self):
 while True:
 full.acquire()
 mutex.acquire()

 item = buffer.popleft()
 print(f"Consumed {item} | Buffer:
{list(buffer)}")

 mutex.release()
 empty.release()
 time.sleep(random.random())

# Create threads
producer = Producer()
consumer = Consumer()

# Start threads
producer.start()
consumer.start()

# Wait for keyboard interrupt
try:
 while True:
 time.sleep(1)
except KeyboardInterrupt:
 print("\nStopping...")
 producer.join()
 consumer.join()
```

3) In producer-consumer problem what difference will it make if we utilize stack for the buffer rather than an array?

1. Default Buffer Structure (Queue - FIFO)
   - Typically, the producer-consumer problem is implemented using a queue (array or linked list) where the producer inserts items at the rear and the consumer removes items from the front.
   - This follows a First-In-First-Out (FIFO) order.
   - Example: If items are produced in order [A, B, C], they will be consumed in the same order: A → B → C.
2. Using a Stack Instead of a Queue
   - A stack follows a Last-In-First-Out (LIFO) order.
   - The most recently produced item is consumed first.
   - Example: If items are produced in order [A, B, C], they will be consumed in the reverse order: C → B → A.
3. Real-World Implications
   - Queue (FIFO) is better when older data needs to be processed first (e.g., task scheduling, message queues).
   - Stack (LIFO) is better for cases where the latest data is most relevant (e.g., browser history, function calls, or stock market trades).
4. Impact on Producer-Consumer Synchronization
   - Both require synchronization (mutex + semaphore) to prevent race conditions.
   - However, with a stack, there might be less predictable behavior in scenarios where ordering matters.