Experiment -4 (OS Lab)
Name: Veeransh Shah
Reg Id: 221070063

Aim:
Write a Python program to simulate producer-consumer problem using semaphores.

Theory:

The producer-consumer problem is a classic synchronisation problem in computer science, where there are two types of processes: producers, which produce items, and consumers, which consume items. The problem arises when multiple producers and consumers are working concurrently, and they need to access a shared buffer. Producers must ensure that the buffer does not overflow, and consumers must ensure that the buffer does not underflow.

Semaphores are a synchronisation primitive used for controlling access to shared resources. They are essentially counters that can be incremented or decremented atomically. In Python, you can use the `threading` module to implement semaphores.

In this program:

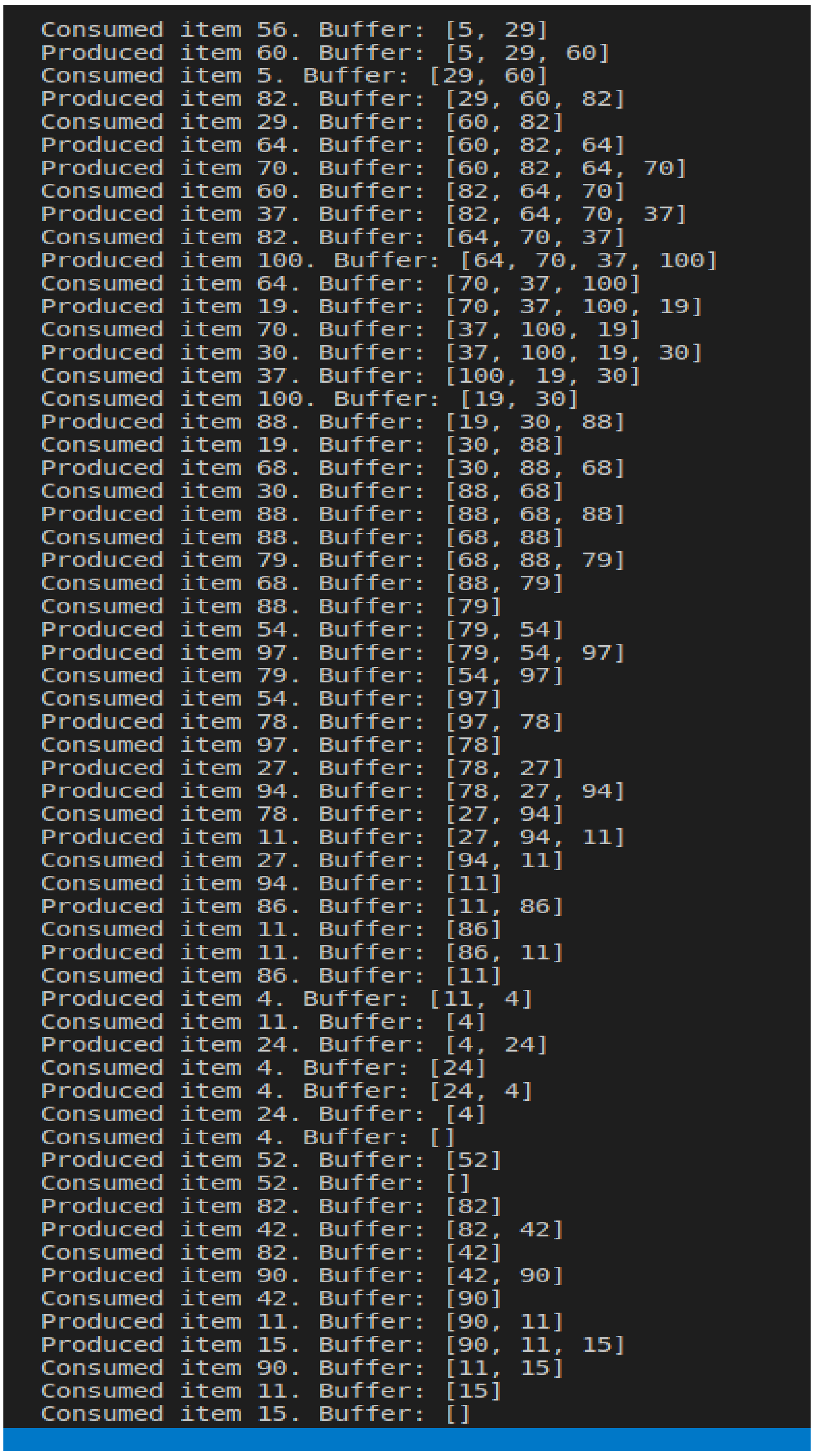
- We define a shared buffer `buffer` with a maximum size of `MAX_BUFFER_SIZE`.
- We use three semaphores: `mutex` for mutual exclusion, `full_sem` to track the number of full slots in the buffer, and `empty_sem` to track the number of empty slots in the buffer.
- The `Producer` class represents a producer thread that generates random items and adds them to the buffer.
- The `Consumer` class represents a consumer thread that removes items from the buffer.
- Both producer and consumer threads acquire and release semaphores to ensure mutual exclusion and to prevent buffer overflow or underflow.
- We create and start multiple producer and consumer threads and then wait for them to finish using the `join()` method.

This program demonstrates how semaphores can be used to synchronise producer and consumer threads in a multi-threaded environment.

Code:
```
import threading
import time
import random

BUFFER_SIZE = 5
mutex = threading.Semaphore(1)
empty = threading.Semaphore(BUFFER_SIZE)
full = threading.Semaphore(0)
```

```python
buffer = []

def producer():
    while True:
        item = random.randint(1, 100)
        empty.acquire()
        mutex.acquire()
        buffer.append(item)
        print(f"Produced item {item}. Buffer: {buffer}")
        mutex.release()
        full.release()
        time.sleep(random.uniform(0.1, 0.5))

def consumer():
    while True:
        full.acquire()
        mutex.acquire()
        item = buffer.pop(0)
        print(f"Consumed item {item}. Buffer: {buffer}")
        mutex.release()
        empty.release()
        time.sleep(random.uniform(0.1, 0.5))

producer_thread = threading.Thread(target=producer)
consumer_thread = threading.Thread(target=consumer)

producer_thread.start()
consumer_thread.start()

producer_thread.join()
consumer_thread.join()

# This program simulates between the producer-consumer problem using
semaphores.
# It creates 2 threads, each for the producer and for the consumer.
# the producer thread randomly and continuously generates random items
and adds them to a shared buffer.
# the consumer thread continuously consumes items from the buffer.
Semaphores are used to control access to the buffer.
```

Output:


```
Consumed item 56. Buffer: [5, 29]
Produced item 60. Buffer: [5, 29, 60]
Consumed item 5. Buffer: [29, 60]
Produced item 82. Buffer: [29, 60, 82]
Consumed item 29. Buffer: [60, 82]
Produced item 64. Buffer: [60, 82, 64]
Produced item 70. Buffer: [60, 82, 64, 70]
Consumed item 60. Buffer: [82, 64, 70]
Produced item 37. Buffer: [82, 64, 70, 37]
Consumed item 82. Buffer: [64, 70, 37]
Produced item 100. Buffer: [64, 70, 37, 100]
Consumed item 64. Buffer: [70, 37, 100]
Produced item 19. Buffer: [70, 37, 100, 19]
Consumed item 70. Buffer: [37, 100, 19]
Produced item 30. Buffer: [37, 100, 19, 30]
Consumed item 37. Buffer: [100, 19, 30]
Consumed item 100. Buffer: [19, 30]
Produced item 88. Buffer: [19, 30, 88]
Consumed item 19. Buffer: [30, 88]
Produced item 68. Buffer: [30, 88, 68]
Consumed item 30. Buffer: [88, 68]
Produced item 88. Buffer: [88, 68, 88]
Consumed item 88. Buffer: [68, 88]
Produced item 79. Buffer: [68, 88, 79]
Consumed item 68. Buffer: [88, 79]
Consumed item 88. Buffer: [79]
Produced item 54. Buffer: [79, 54]
Produced item 97. Buffer: [79, 54, 97]
Consumed item 79. Buffer: [54, 97]
Consumed item 54. Buffer: [97]
Produced item 78. Buffer: [97, 78]
Consumed item 97. Buffer: [78]
Produced item 27. Buffer: [78, 27]
Produced item 94. Buffer: [78, 27, 94]
Consumed item 78. Buffer: [27, 94]
Produced item 11. Buffer: [27, 94, 11]
Consumed item 27. Buffer: [94, 11]
Consumed item 94. Buffer: [11]
Produced item 86. Buffer: [11, 86]
Consumed item 11. Buffer: [86]
Produced item 11. Buffer: [86, 11]
Consumed item 86. Buffer: [11]
Produced item 4. Buffer: [11, 4]
Consumed item 11. Buffer: [4]
Produced item 24. Buffer: [4, 24]
Consumed item 4. Buffer: [24]
Produced item 4. Buffer: [24, 4]
Consumed item 24. Buffer: [4]
Consumed item 4. Buffer: []
Produced item 52. Buffer: [52]
Consumed item 52. Buffer: []
Produced item 82. Buffer: [82]
Produced item 42. Buffer: [82, 42]
Consumed item 82. Buffer: [42]
Produced item 90. Buffer: [42, 90]
Consumed item 42. Buffer: [90]
Produced item 11. Buffer: [90, 11]
Produced item 15. Buffer: [90, 11, 15]
Consumed item 90. Buffer: [11, 15]
Consumed item 11. Buffer: [15]
Consumed item 15. Buffer: []
```

Conclusion:
The producer-consumer problem represents a common synchronisation challenge in concurrent programming, where multiple producers generate data and multiple consumers process it concurrently. This problem highlights the need for effective coordination to prevent issues such as buffer overflow or underflow.