VIDHI ROHIRA

S.Y B.TECH

SEM III

COMPUTER ENGINEERING

OS LAB 4

231071052

BATCH – C

LABORATORY 4

AIM:- To implement semaphores and solve producer consumer problem using semaphores.

THEORY:-

Q] WHAT ARE SEMAPHORES?

A semaphore is a synchronization mechanism used to control access to shared resources in a concurrent or multi-process environment. It helps in preventing race conditions, where two or more processes or threads try to access shared resources at the same time, potentially leading to incorrect or inconsistent results. Semaphores play a key role in process synchronization in operating systems, allowing processes to wait for specific conditions before proceeding. In the wait operation, a process checks if the semaphore's value is greater than 0 (indicating resource availability). If it is, the value is decremented by 1, allowing the process to access the resource. If the value is 0, the process is forced to wait until the resource becomes available. The signal operation is the counterpart to wait. When a process finishes using a resource, it increments the semaphore's value by 1, signalling that the resource is now available for other processes. This allows any waiting processes to proceed.

Q] ADVANTAGES AND DISADVANTAGES OF SEMAPHORES.

Advantages of Semaphores

1. **Efficient Resource Management**: Semaphores control access to shared resources, enabling smooth concurrent execution.
2. **Simplifies Synchronization**: They handle synchronization in problems like producer-consumer, ensuring safe access to shared resources.
3. **Flexibility**: Semaphores work well in multithreading scenarios, managing varying thread numbers.
4. **Prevents Deadlock (if used correctly)**: Can avoid deadlock by carefully controlling resource access.

Disadvantages of Semaphores

1. **Risk of Deadlock and Starvation**: Incorrect use can cause deadlock (infinite waiting) or starvation (threads never getting resources).
2. **Complexity in Large Systems**: Managing many semaphores can become complex and prone to errors.
3. **Busy Waiting**: Some implementations can waste CPU time when threads repeatedly check semaphores.
4. **No Ownership Enforcement**: Threads can mistakenly release semaphores they didn't acquire, causing synchronization issues.

Q] WHAT IS A PRODUCER CONSUMER PROBLEM?

The Producer-Consumer problem is a classic synchronization problem that illustrates the need for process cooperation. It involves two types of processes: producers and consumers, which share a common, finite buffer. The producer is responsible for generating items and placing them into the buffer, while the consumer retrieves these items for processing. The challenge arises when the buffer becomes full or empty, requiring proper coordination between producers and consumers to prevent overwriting data or consuming non-existent data.

In the Producer-Consumer problem, the buffer serves as a shared resource where the producer stores the items and the consumer removes them. The buffer has a limited size, which means that a producer cannot add items if the buffer is full, and a consumer cannot consume items if the buffer is empty. Without proper synchronization, the producer and consumer could access the buffer simultaneously, leading to race conditions where multiple processes corrupt the data or overwrite each other’s work.

Q] HOW CAN WE SOLVE PRODUCER-CONSUMER PROBLEM USING SEMAPHORES?

To solve the Producer-Consumer problem using semaphores, we need to implement a synchronization mechanism that ensures proper coordination between the producer and consumer processes. The solution involves using three main components: a shared buffer, semaphores to manage access to the buffer, and a mutual exclusion mechanism to ensure that only one process accesses the buffer at any given time.

ALGORITHM

1. **Initialize Semaphores and Shared Variables**
   - **Input**:
     - item_count: Total items to produce and consume.
     - BUFFER_SIZE: Size of the shared buffer.
   - **Process**:
     - Create a buffer of size BUFFER_SIZE.
     - Initialize semaphore empty_slots with a count of BUFFER_SIZE to track empty slots.
     - Initialize semaphore full_slots with a count of 0 to track filled slots.
     - Initialize semaphore buffer_mutex with a count of 1 to ensure exclusive access to the buffer.

2. **Producer Function**
   - **Input**: item_count (number of items to produce).
   - **Process** (for each item):
     - Call empty_slots.wait() to wait for an empty slot in the buffer.
     - Call buffer_mutex.wait() to gain exclusive access to the buffer.
     - Add the item to the buffer at in_index.
     - Print item and position.
     - Update in_index for circular buffer indexing.
     - Call buffer_mutex.signal() to release the buffer lock.
     - Call full_slots.signal() to signal that a new slot is filled.
   - **Output**: Produced items in the buffer.

3. **Consumer Function**
   - **Input**: item_count (number of items to consume).
   - **Process** (for each item):
     - Call full_slots.wait() to wait for a filled slot in the buffer.
     - Call buffer_mutex.wait() to gain exclusive access to the buffer.
     - Remove an item from the buffer at out_index.
     - Print item and position.
     - Update out_index for circular buffer indexing.
     - Call buffer_mutex.signal() to release the buffer lock.
     - Call empty_slots.signal() to signal that a new slot is empty.
   - **Output**: Consumed items from the buffer.

4. **Main Execution**
    - **Process**:
        - Start producer_thread and consumer_thread.
        - Wait for both threads to complete.
    - **Output**: Completed producer and consumer operations.

CODE:

```
import threading
import time

class Semaphore:
    def __init__(self, initial):
        self.count = initial
        self.lock = threading.Lock()  # Lock to control access to the count

    def wait(self):
        # Decrement the semaphore count if it's positive
        while True:
            with self.lock:
                if self.count > 0:
                    self.count -= 1
                    break

    def signal(self):
        # Increment the semaphore count
        with self.lock:
            self.count += 1

def producer(item_count):
    global in_index, buffer
    for item in range(item_count):
        empty_slots.wait()  # Wait for an empty slot in the buffer
        buffer_mutex.wait()  # Acquire lock for exclusive buffer access
        # Add item to the buffer
        buffer[in_index] = item
        print(f"Produced item: {item} at position {in_index}")
        in_index = (in_index + 1) % BUFFER_SIZE  # Circular increment for
index
        buffer_mutex.signal()  # Release buffer lock
        full_slots.signal()  # Signal that a new slot is filled
        time.sleep(0.1)  # Simulate time taken for production
```

```python
def consumer(item_count):
    global out_index, buffer
    for _ in range(item_count):
        full_slots.wait()  # Wait for a filled slot in the buffer
        buffer_mutex.wait()  # Acquire lock for exclusive buffer access
        # Retrieve item from the buffer
        item = buffer[out_index]
        print(f"Consumed item: {item} from position {out_index}")
        out_index = (out_index + 1) % BUFFER_SIZE  # Circular increment for index
        buffer_mutex.signal()  # Release buffer lock
        empty_slots.signal()  # Signal that a new slot is empty
        time.sleep(0.1)  # Simulate time taken for consumption

def main():
    global BUFFER_SIZE, buffer, in_index, out_index, empty_slots, full_slots, buffer_mutex
    # Take user inputs for item count and buffer size
    item_count = int(input("Enter the number of items to produce and consume: "))
    BUFFER_SIZE = int(input("Enter the buffer size: "))

    # Initialize buffer and semaphores based on user input
    buffer = [None] * BUFFER_SIZE
    in_index = 0
    out_index = 0
    empty_slots = Semaphore(BUFFER_SIZE)
    full_slots = Semaphore(0)
    buffer_mutex = Semaphore(1)

    # Create and start producer and consumer threads
    producer_thread = threading.Thread(target=producer, args=(item_count,))
    consumer_thread = threading.Thread(target=consumer, args=(item_count,))

    producer_thread.start()  # Start producer thread
    consumer_thread.start()  # Start consumer thread

    producer_thread.join()  # Wait for producer thread to complete
    consumer_thread.join()  # Wait for consumer thread to complete

if __name__ == "__main__":
    main()
```

OUTPUT:-

```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/OS LAB/semaphores.py"
Enter the number of items to produce and consume: 6
Enter the buffer size: 3
Produced item: 0 at position 0
Consumed item: 0 from position 0
Produced item: 1 at position 1
Consumed item: 1 from position 1
Produced item: 2 at position 2
Consumed item: 2 from position 2
Produced item: 3 at position 0
Consumed item: 3 from position 0
Produced item: 4 at position 1
Consumed item: 4 from position 1
Produced item: 5 at position 2
Consumed item: 5 from position 2
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB>
```

CONCLUSION:-

In this lab session, we learned about semaphores and their application in solving the producer-consumer problem. Semaphores are synchronization mechanisms that help manage concurrent access to shared resources by using wait and signal operations. In the producer-consumer problem, a producer generates items for a shared buffer, and a consumer retrieves items from it, necessitating careful coordination to avoid overwriting or prematurely consuming items. We implemented a solution using three semaphores in Python: empty_slots to track available space, full_slots to count filled slots, and buffer_mutex to ensure exclusive buffer access. This approach allowed the producer and consumer to operate in sync, illustrating the role of semaphores in effective resource management and concurrency control.