**Lab no 8**

**Producer-consumer Problem using semaphores**

***Objectives:***

* What is Producer-Consumer problem?
* Implementation of producer-consumer problem.

**Producer Consumer Problem:**

The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e. synchronization between more than one processes.

In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size.

**What's the problem here?**

The following are the problems that might occur in the Producer-Consumer:

* The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.
* The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
* The producer and consumer should not access the buffer at the same time.

**What's the solution?**

The above three problems can be solved with the help of semaphores(learn more about semaphores from [here](https://afteracademy.com/blog/what-is-semaphore-and-what-are-its-types)).

In the producer-consumer problem, we use three semaphore variables:

**Producer-Consumer Problem** consists of 3 components:

### ****Bounded Buffer****

* A buffer is temporary storage that is accessible by different threads. A simple example of a buffer is an array. Multiple threads can read the data from the buffer as well as can write the data to the buffer concurrently. A bounded buffer is one that has a limited capacity and can’t store the data beyond its capacity.

### . ****Producer Thread****

* A Producer Thread is one that generates some data, puts it into the buffer, and starts again until all the data needed is not produced. An example of this could be a thread that downloads some data over the network and stores it temporarily into the buffer

### .  ****Consumer Thread****

* A Consumer Thread is one that consumes the data present inside the buffer, uses it for some task, and starts again until the task assigned to the thread is not completed. An example of this could be a thread that reads the data that is downloaded over the internet and stores it in the database.

The code can be summarized as:

1. **empty:** This semaphore stores the number of slots that are empty in our buffer. The initial value of this semaphore is the size of our bounded buffer. Before adding any data in the buffer, the Producer thread will try to acquire this semaphore and will decrease its value by 1. If the value of this semaphore is already 0, this means that the buffer is full and our empty semaphore will block the Producer Thread until the value of the empty semaphore becomes greater than 0. Similarly, after the Consumer Thread has consumed the data from the buffer, it will release this semaphore, increasing the value of the semaphore by 1.
2. **full:** This semaphore stores the number of slots that are full in our buffer. The initial value of this semaphore is 0. Before consuming the data from the buffer, the Consumer Thread will try to acquire this semaphore. If the value of this semaphore is already 0, this means that the buffer is already empty and our full semaphore will block the Consumer Thread until the value of the full semaphore becomes greater than 0. Similarly, the Producer Thread will release this semaphore after it has added one item in it.
3. **mutex:** This semaphore will handle the race condition by allowing only one semaphore to operate on the shared buffer at a time. The initial value of this semaphore is 1. Before operating on the shared buffer, both threads will try to acquire this semaphore. If any thread found the value of this semaphore as 0, this means that the other thread is operating on the buffer and it will be blocked by the semaphore. After operating on the buffer, the working thread will release this semaphore so that the other thread can operate on the buffer.

We also maintain 2 pointer to help our threads where to add or take the data.

* **in pointer:**

This pointer will tell our Producer Thread where to add the next data in the buffer generated by the producer. After adding, the pointer is incremented by 1.

* **out pointer:**

This pointer will tell our Consumer Thread where to read the next data from the buffer. After reading, the pointer is incremented by 1.

Example :

Let us check the implementation on how to solve this problem in Python. Say we have a bounded buffer of capacity 10. The Producer Thread will produce 20 items and the Consumer Thread will consume those 20 items produced by the Producer. Adding time.sleep(1) in Producer and time.sleep(2.5) in Consumer makes our Producer Thread operate faster than Consumer Thread. Even if we are starting our Consumer Thread first, it will wait till there is no data present in our buffer.

CODE :

import threading

import time

# Shared Memory variables

CAPACITY = 10

buffer = [-1 for i in range(CAPACITY)]

in\_index = 0

out\_index = 0

# Declaring Semaphores

mutex = threading.Semaphore()

empty = threading.Semaphore(CAPACITY)

full = threading.Semaphore(0)

# Producer Thread Class

class Producer(threading.Thread):

  def run(self):

    global CAPACITY, buffer, in\_index, out\_index

    global mutex, empty, full

    items\_produced = 0

    counter = 0

    while items\_produced < 20:

      empty.acquire()

      mutex.acquire()

      counter += 1

      buffer[in\_index] = counter

      in\_index = (in\_index + 1)%CAPACITY

      print("Producer produced : ", counter)

      mutex.release()

      full.release()

      time.sleep(1)

      items\_produced += 1

# Consumer Thread Class

class Consumer(threading.Thread):

  def run(self):

    global CAPACITY, buffer, in\_index, out\_index, counter

    global mutex, empty, full

    items\_consumed = 0

    while items\_consumed < 20:

      full.acquire()

      mutex.acquire()

      item = buffer[out\_index]

      out\_index = (out\_index + 1)%CAPACITY

      print("Consumer consumed item : ", item)

      mutex.release()

      empty.release()

      time.sleep(2.5)

      items\_consumed += 1

# Creating Threads

producer = Producer()

consumer = Consumer()

# Starting Threads

consumer.start()

producer.start()

# Waiting for threads to complete

producer.join()

consumer.join()

**Task:**

* Execute the above Producer Consumer Problem and show the output.

### Text Description automatically generated with medium confidence