**Unit 4: The Role of the Producer-Consumer Pattern in Facilitating Quality of Service in Distributed Applications**

Welcome to Week 4. This week, you will be introduced to examples of code that may assist with two core aspects of your assessment – the Producer-Consumer mechanism and Quality of Service. The Producer-Consumer mechanism supports threading so that multiple requests can be responded to in an organized way. Quality of Service is an important concept when supporting distributed communications, ensuring that the performance requirements of customers are fulfilled within a dynamically changing network.

**On completion of this unit, you will be able to:**

* Create a producer-consumer implementation using Python.
* Research the literature to identify the Quality-of-Service requirements of distributed applications.

**Reflection:**

QoS criteria are critical parameters that outline the level of performance, availability, scalability, and maintainability that must be met by a system. Needs from the business are what drive the quality-of-service specifications.

As global infrastructure evolves to accommodate new technologies, more and more organizations are opting for distributed systems. This subfield of computer science encompasses a wide range of topics. Users perceive the distributed system as a single machine, but it consists of many independent computers working together. The term "quality of service" is used to describe the processing of data traffic on a network with the goal of minimizing packet loss, delay, and jitter. (Bruneo et al., 2013)

**Code for producer consumer implementation (**askpython., 2021)**:**

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()

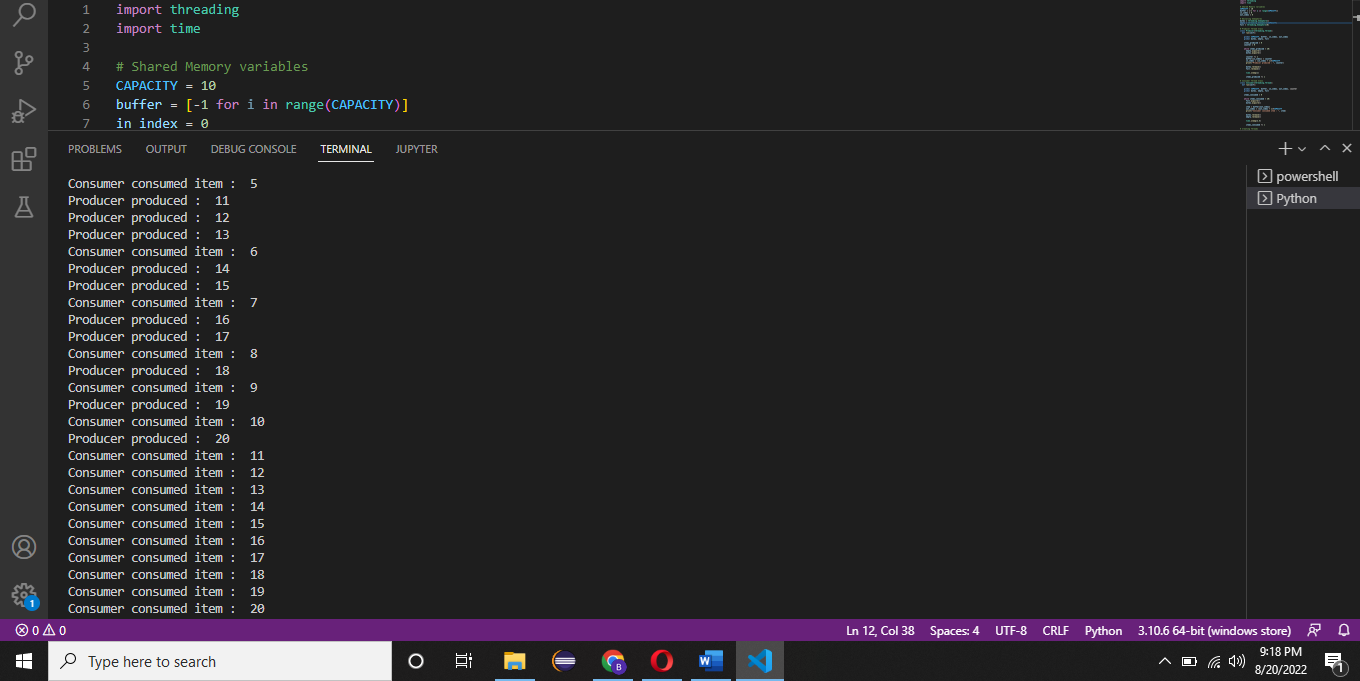
producer.start()

# Waiting for threads to complete

producer.join()

consumer.join()

**Output:**



**References:**

Bruneo, D., Distefano, S., Longo, F. and Scarpa, M. (2013). Stochastic Evaluation of QoS in Service-Based Systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(10), pp.2090–2099. doi:10.1109/tpds.2012.313.

Askpython. (2021, June 28). Producer-Consumer Problem in Python - AskPython. Retrieved August 20, 2022, from askpython.com website: https://www.askpython.com/python/producer-consumer-problem