# 1 Assignment 3

1. Implement Producer-Consumer problem (PCP). Analyze the significance of semaphore, mutex, bounded buffer, producer thread, consumer thread using the code available on Producer- Consumer Problem in Python - AskPython. (a) Write a brief about the problem and solution. (b) Code and Output
2. Demonstrate how PCP occurs for a application of your choice.

Ans >

The producer-consumer problem is a synchronization challenge in the field of operating systems, particularly in scenarios involving concurrent programming and multi-threading. It revolves around two types of processes:

- Producers: These processes are responsible for generating data or items and placing them in a shared buffer.
- Consumers: These processes retrieve and consume items from the buffer.

---

The primary goal is to ensure the following conditions are met:

- Producers should refrain from producing items if the buffer is full.
- Consumers should avoid consuming items if the buffer is empty.
- The central objective is to maintain synchronization between producers and consumers to prevent issues such as data corruption, race conditions, and deadlocks.

---

Solution Approach:

- Shared Buffer:
  - A fixed-size buffer is utilized, acting as a common storage space for both producers and consumers.
- Semaphore for Empty Slots (empty):
  - Initialized to the size of the buffer.Represents the count of empty slots in the buffer. Decreases by producers when they add an item. Decreases by consumers when they remove an item.
- Semaphore for Full Slots (full):
  - Initialized to 0. Represents the count of filled slots in the buffer. Increased by producers when they add an item. Decreases by consumers when they remove an

item.

- Illustration with a Different Example:
    - Let's consider a scenario in a restaurant where there are chefs (producers) preparing dishes and waiters (consumers) serving these dishes to customers. The shared buffer is the kitchen counter where dishes are temporarily placed before being served.

---

- Shared Buffer (Kitchen Counter):
    - Represents the kitchen counter where the prepared dishes are temporarily stored.
- Semaphore for Empty Serving Plates (empty):
    - Initialized to the maximum capacity of the counter. Indicates the count of empty serving plates on the kitchen counter. Decreases as chefs place prepared dishes on empty plates. Decreases as waiters take dishes from the counter to serve customers.
- Semaphore for Full Serving Plates (full):
    - Initialized to 0. Represents the count of plates with prepared dishes on the counter. Increases as chefs place dishes on empty plates. Decreases as waiters take dishes from the counter to serve customers.
- In this analogy, the restaurant ensures that chefs don't prepare more dishes if there are no empty plates, and waiters don't serve if there are no prepared dishes on the counter, effectively managing the flow of food production and service.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

```

In [2]:
```
import threading
import time

```

In [3]:
```
# Shared Memory variables
CAPACITY = 10
buffer = [-1 for i in range(CAPACITY)]
in_index = 0
out_index = 0
```

In [4]:
```
# Declaring Semaphores
mutex = threading.Semaphore()
empty = threading.Semaphore(CAPACITY)
full = threading.Semaphore(0)
```

In [5]:
```python
# 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(0)
   items_produced += 1

```

In [6]:
```python
# 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(0.5)
   items_consumed += 1
```

In [7]:
```
producer = Producer()
consumer = Consumer()
consumer.start()
producer.start()
producer.join()
consumer.join()
```

```
Producer produced :  1
Producer produced :  2
Producer produced :  3
Producer produced :  4
Producer produced :  5
Producer produced :  6
Producer produced :  7
Producer produced :  8
Producer produced :  9
Producer produced :  10
Consumer consumed item :  1
Producer produced :  11
Consumer consumed item :  2
Producer produced :  12
Consumer consumed item :  3
Producer produced :  13
Consumer consumed item :  4
Producer produced :  14
Consumer consumed item :  5
Producer produced :  15
Consumer consumed item :  6
Producer produced :  16
Consumer consumed item :  7
Producer produced :  17
Consumer consumed item :  8
Producer produced :  18
Consumer consumed item :  9
Producer produced :  19
Consumer consumed item :  10
Producer produced :  20
Consumer consumed item :  11
Consumer consumed item :  12
Consumer consumed item :  13
Consumer consumed item :  14
Consumer consumed item :  15
Consumer consumed item :  16
Consumer consumed item :  17
Consumer consumed item :  18
Consumer consumed item :  19
Consumer consumed item :  20
```

In [8]:
```
import time
```

In [9]:
```
CAPACITY = 10
buffer = [-1 for i in range(CAPACITY)]
in_index = 0
out_index = 0
```

In [10]:
```
mutex = threading.Semaphore()
empty = threading.Semaphore(CAPACITY)
full = threading.Semaphore(0)

```

In [11]:
```
class Producer(threading.Thread):
 def run(self):
  global CAPACITY, buffer, in_index
  global mutex, empty, full
  for counter in range(1, 21):
   empty.acquire()
   mutex.acquire()
   buffer[in_index] = counter
   in_index = (in_index + 1) % CAPACITY
   print("Producer produced:", counter)
   mutex.release()
   full.release()
   time.sleep(0)

```

In [12]:
```
class Consumer(threading.Thread):
 def run(self):
  global CAPACITY, buffer, out_index
  global mutex, empty, full
  for _ in range(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(0.5)
```

In [13]:
```
producer = Producer()
consumer = Consumer()
consumer.start()
producer.start()
producer.join()
consumer.join()
```

```
Producer produced: 1
Producer produced: 2
Producer produced: 3
Producer produced: 4
Producer produced: 5
Producer produced: 6
Producer produced: 7
Producer produced: 8
Producer produced: 9
Producer produced: 10
Consumer consumed item: 1
Producer produced: 11
Consumer consumed item: 2
Producer produced: 12
Consumer consumed item: 3
Producer produced: 13
Consumer consumed item: 4
Producer produced: 14
Consumer consumed item: 5
Producer produced: 15
Consumer consumed item: 6
Producer produced: 16
Consumer consumed item: 7
Producer produced: 17
Consumer consumed item: 8
Producer produced: 18
Consumer consumed item: 9
Producer produced: 19
Consumer consumed item: 10
Producer produced: 20
Consumer consumed item: 11
Consumer consumed item: 12
Consumer consumed item: 13
Consumer consumed item: 14
Consumer consumed item: 15
Consumer consumed item: 16
Consumer consumed item: 17
Consumer consumed item: 18
Consumer consumed item: 19
Consumer consumed item: 20
```

In [14]:
```
import queue
import random
```

In [15]:
```
MAX_QUEUE_SIZE = 5
event_queue = queue.Queue(MAX_QUEUE_SIZE)
mutex = threading.Lock()
empty = threading.Semaphore(MAX_QUEUE_SIZE)
full = threading.Semaphore(0)
```

In [16]:
```
class UserClickProducer(threading.Thread):
 def run(self):
  global MAX_QUEUE_SIZE, event_queue
  global mutex, empty, full
  for _ in range(10):
   print("User clicked")
   empty.acquire()
   mutex.acquire()
   event_queue.put("Click")
   mutex.release()
   full.release()
   time.sleep(random.uniform(0.1, 0.5))

```

In [17]:
```
class EventHandlerConsumer(threading.Thread):
 def run(self):
  global MAX_QUEUE_SIZE, event_queue
  global mutex, empty, full
  for _ in range(10):
   full.acquire()
   mutex.acquire()
   event = event_queue.get()
   print(f"Handling event: {event}")
   mutex.release()
   empty.release()
   time.sleep(random.uniform(0.1, 0.5))

```

In [18]:
```
user_click_producer = UserClickProducer()
event_handler_consumer = EventHandlerConsumer()
user_click_producer.start()
event_handler_consumer.start()
user_click_producer.join()
event_handler_consumer.join()

```

```
User clicked
Handling event: Click
User clicked
Handling event: Click
User clicked
Handling event: Click
User clicked
Handling event: Click
User clicked
Handling event: Click
User clicked
Handling event: Click
User clicked
Handling event: Click
User clicked
User clicked
Handling event: Click
User clicked
Handling event: Click
Handling event: Click
```

In [19]:
```
import queue
import random
```

Example

- Event Handling in GUI:
  - Producers: User input events (clicks, keystrokes).
  - Consumers: Event handlers or listeners.
  - Buffer: Event queue.

In [20]:
```
MAX_QUEUE_SIZE = 5
event_queue = queue.Queue(MAX_QUEUE_SIZE)
mutex = threading.Lock()
empty = threading.Semaphore(MAX_QUEUE_SIZE)
full = threading.Semaphore(0)
```

In [21]:
```
class UserClickProducer(threading.Thread):
    def run(self):
        global MAX_QUEUE_SIZE, event_queue
        global mutex, empty, full
        for _ in range(10):
            print("User clicked")
empty.acquire()
mutex.acquire()
event_queue.put("Click")
mutex.release()
full.release()
time.sleep(random.uniform(0.1, 0.5))
```

In [22]:
```
class EventHandlerConsumer(threading.Thread):
    def run(self):
        global MAX_QUEUE_SIZE, event_queue
        global mutex, empty, full
        for _ in range(10):
            full.acquire()
            mutex.acquire()
            event = event_queue.get()
            print(f"Handling event: {event}")
            mutex.release()
            empty.release()
            time.sleep(random.uniform(0.1, 0.5))
```

In [ ]:
```
user_click_producer = UserClickProducer()
event_handler_consumer = EventHandlerConsumer()
user_click_producer.start()
event_handler_consumer.start()
user_click_producer.join()
event_handler_consumer.join()
```

```
User clicked
User clicked
User clicked
User clicked
User clicked
User clicked
User clicked
User clicked
User clicked
User clicked
Handling event: Click
```

In [ ]:
```

```