SET-224 /CET-225
Operating Systems

LAB # 10

LAB Title

Implementation of Semaphore Mechanism

Assessment of CLO: 04, PLO: 03

Student Name:			
Roll No.			
Semester		Session	

S. No.	Perf. Level / Criteria	Excellent (2.5)	Good (2)	Satisfactory (1.5)	Needs Improvement (0 ~ 1)	Marks Obtained
1	Project Execution & Implementation	Fully functional, optimized, and well-structured.	Minor errors, mostly functional.	Some errors, requires guidance.	Major errors, non-functional, or not Performed.	
2	Results & Debugging Or Troubleshooting	Accurate results with effective debugging Or Troubleshooting.	Mostly correct, some debugging Or Troubleshooting needed.	Partial results, minimal debugging Or Troubleshooting.	Incorrect results, no debugging Or Troubleshooting, or not attempted.	
3	Problem-Solving & Adaptability (VIVA)	Creative approach, efficiently solves challenges.	Adapts well, minor struggles.	Some adaptability, needs guidance.	Lacks innovation or no innovation, unable to solve problems.	
4	Report Quality & Documentation	Clear, structured, with detailed visuals.	Mostly clear, minor gaps.	Some clarity issues, missing details.	Poorly structured, lacks clarity, or not submitted.	
					Total Marks Obtained Out of 10	

Experiment evaluated by

Instructor's Name	Engr.Bushra Aziz		
Date		Signature	

Objective: To be familiar with implementation of semaphore .Solving producer-consumer *(Classical Problem)* problem in Python using semaphores.

Theory:

Semaphores

Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multi-processing environment.

The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1 only. A semaphore can only be accessed using the following operations: wait() and signal().

```
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
          s.value = zero;
      else {
                  /* place this process in s.queue */;
                  /* block this process */;
      }
```

```
void semSignalB(semaphore s)
{
       if (s.queue is empty())
           s.value = one;
       else {
                   /* remove a process P from s.queue */;
                   /* place process P on ready list */;
       }
}
```

[In python, acquire() and release() provide wait() and signal() functionality, respectively]

Python's Simple Lock

using class threading.Lock

A simple mutual exclusion lock used to limit access to one thread. This is a semaphore with s = 1.

acquire()

Obtain a lock. The process is blocked until the lock is available.

release()

Release the lock and if another thread is waiting for the lock, wake up that thread.

Python's Semaphore

using class threading.Semaphore(s)

acquire()

Obtain a semaphore. The process is blocked until the semaphore is available.

release()

Release the semaphore and if another thread is waiting for the semaphore, wake up that thread.

Python: Producer-Consumer Solution using Semaphores

The **Producer-Consumer problem** (also known as the **Bounded Buffer problem**) is a classic **inter-process communication (IPC)** and **synchronization** scenario in **Operating Systems**. It models the coordination between processes (or threads) that produce and consume shared resources.

There are two types of processes:

- Producer: Produces data and stores it in a shared buffer.
- Consumer: Takes data from the buffer and uses it.

The **buffer** has a **limited size** (bounded).

- Producer must **wait** if the buffer is **full**.
- Consumer must **wait** if the buffer is **empty**.

The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value max_c (max_c=10 in this example); the semaphore full is initialized to the value 0.

Sample CODE:

```
from threading import Semaphore

import time

import threading

import random

max_c=10

full = Semaphore(0)

empty =Semaphore(max_c)

mutex=Semaphore(1)

queue = []


def producer(i):

  empty.acquire()

  mutex.acquire()

  print("Produced:",i)

  queue.append(i)

  mutex.release()

  full.release()
```

```
def consumer(i):
  full.acquire()
  mutex.acquire()
  print("consumer has consumed:",queue.pop(0))
  mutex.release()
  empty.release()



total = []
for i in range(max_c):
 tp=threading.Thread(target=producer,args=[i])
 tc=threading.Thread(target=consumer,args=[i])
 total.append(tp)
 total.append(tc)

random.shuffle(total)

for i in total:
  i.start()
  time.sleep(1)
```

Exercise:

Write a python program that demonstrates the synchronization of Readers and Writer Problem using Semaphore.