**Institute of Computer Technology**

**B. Tech Computer Science and Engineering**

**Sub: Operating System (2CSE402)**

**PRACTICAL – 3**

**Synchronization**

**A)   Write a C program to simulate producer-consumer problem using semaphores.**

**Code:**

from threading import Thread, Semaphore

class Printer():

consume = Semaphore(0)

produce = Semaphore(1)

item = 0

def printData(self):

self.consume.acquire()

print('Consumed : ', self.item)

self.produce.release()

def setData(self,item):

self.produce.acquire()

self.item = item

print('Produced : ', self.item)

self.consume.release()

class Producer(Thread):

def \_\_init\_\_(self,printer):

super().\_\_init\_\_()

self.printer = printer

def run(self):

for i in range(20):

self.printer.setData(i)

class Consumer(Thread):

def \_\_init\_\_(self,printer):

super().\_\_init\_\_()

self.printer = printer

def run(self):

for i in range(20):

self.printer.printData()

printer = Printer()

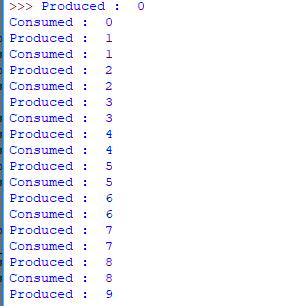
consumer = Consumer(printer)

producer = Producer(printer)

consumer.start()

producer.start()

**Output:**



**B)** **Write a C program to simulate the concept of Dining-Philosophers problem.**

**Code:**

import threading

import random

import time

class Philosopher(threading.Thread):

running = True

def \_\_init\_\_(self, xname, forkOnLeft, forkOnRight):

threading.Thread.\_\_init\_\_(self)

self.name = xname

self.forkOnLeft = forkOnLeft

self.forkOnRight = forkOnRight

def run(self):

while (self.running):

# Philosopher is thinking (but really is sleeping).

time.sleep(random.uniform(3, 13))

print('%s is hungry.' % self.name)

self.dine()

def dine(self):

fork1, fork2 = self.forkOnLeft, self.forkOnRight

while self.running:

fork1.acquire(True)

locked = fork2.acquire(False)

if locked: break

fork1.release()

print

'%s swaps forks' % self.name

fork1, fork2 = fork2, fork1

else:

return

self.dining()

fork2.release()

fork1.release()

def dining(self):

print('%s starts eating ' % self.name)

time.sleep(random.uniform(1, 10))

print('%s finishes eating and leaves to think.' % self.name)

def DiningPhilosophers():

forks = [threading.Lock() for n in range(5)]

philosopherNames = ('Aristotle', 'Kant', 'Buddha', 'Marx', 'Russel')

philosophers = [Philosopher(philosopherNames[i], forks[i % 5], forks[(i + 1) % 5]) \

for i in range(5)]

random.seed(507129)

Philosopher.running = True

for p in philosophers: p.start()

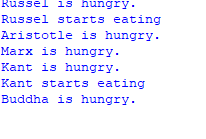
time.sleep(100)

Philosopher.running = False

print("Now we're finishing.")

DiningPhilosophers()

**Output:**



**C) Write A Program to simulate Reader-Writers Problem**

**CODE:**

from threading import Lock, Thread

class ReaderWriter:

rw\_lock = Lock()

lock = Lock()

read\_count = 0

def write(self, value):

self.rw\_lock.acquire()

print('Writing :', value)

self.rw\_lock.release()

def read(self,value):

self.lock.acquire()

self.read\_count+=1

if self.read\_count is 1:

self.rw\_lock.acquire()

self.lock.release()

print('Reading :',value)

self.lock.acquire()

self.read\_count-=1

if self.read\_count is 0:

self.rw\_lock.release()

self.lock.release()

class Reader(Thread):

def \_\_init\_\_(self,rw):

super().\_\_init\_\_()

self.rw = rw

def run(self):

for i in range(20):

self.rw.read(i)

class Writer(Thread):

def \_\_init\_\_(self,rw):

super().\_\_init\_\_()

self.rw = rw

def run(self):

for i in range(20):

self.rw.write(i)

rw = ReaderWriter()

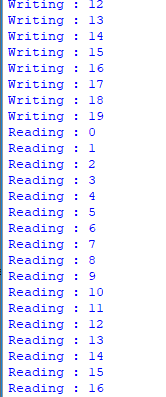
reader = Reader(rw)

writer = Writer(rw)

writer.start()

reader.start()

**OUTPUT:**



**PRE-LAB QUESTIONS**

1. What is the need for process synchronization?

A) Process synchronization is the task of synchronizing the execution of processes in such a manner that no two processes have access to the same shared data and resource.

2. Define a semaphore?

A) 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.

3. Define producer-consumer problem?

A) In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

* The producer’s job is to generate data, put it into the buffer, and start again.
* At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

**Problem**  
To make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer.

**POST-LAB QUESTIONS**

1. Discuss the consequences of considering bounded and unbounded buffers in producer-consumer problem?

A)

Bounded Buffer:

1 producer, N consumers.

It cannot access the shared memory concurrently.

Unbounded Buffer:

It can access the shared memory concurrently.

N producers, N consumers

2. Can producer and consumer processes access the shared memory concurrently? If not which technique provides such a benefit?

A) No.

This problem can be solved by Peterson’s Solution .