18CSC207J-Advance Programming Practice - Structured Programming – Lab Programs

Lab 6 – Parallel and Concurrent Programming Paradigm

Name :- Puneet Sharma
Reg. No. :- RA1911003010331
Class :-CSE F1

Q. Assume we have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.
To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.
Solution:

```
import threading
import random
import time

q_331 = []
empty_331 = threading.Semaphore(1)
full_331 = threading.Semaphore(0)

def producer_331():
    nums = range(5)
    global q_331
    while True:
        no_331 =int(randint(1,100))

        empty_331.acquire()

        q_331.append(no_331)
        print("Produced", no_331)

        full_331.release()

        time.sleep(random.randrange(0, 3))

def consumer_331():
    global q_331
    while True:
        full_331.acquire()

        no_331 = q_331.pop(0)
        print("Consumed", no_331)
```

```
        empty_331.release()

        time.sleep(random.randrange(0, 3))

producerThread = threading.Thread(target=producer_331)
consumerThread = threading.Thread(target=consumer_331)

producerThread.start()
consumerThread.start()
```

Output:

```
In [22]: import threading
         import random
         import time

         q_331 = []
         empty_331 = threading.Semaphore(1)
         full_331 = threading.Semaphore(0)

         def producer_331():
             nums = range(5)
             global q_331
             while True:
                 no_331 =int(randint(1,100))

                 empty_331.acquire()

                 q_331.append(no_331)
                 print("Produced", no_331)

                 full_331.release()

                 time.sleep(random.randrange(0, 3))

         def consumer_331():
             global q_331
             while True:
                 full_331.acquire()

                 no_331 = q_331.pop(0)
                 print("Consumed", no_331)

                 empty_331.release()

                 time.sleep(random.randrange(0, 3))

         producerThread = threading.Thread(target=producer_331)
         consumerThread = threading.Thread(target=consumer_331)

         producerThread.start()
         consumerThread.start()
```

```
Produced 30
Consumed 30
Produced 40
Consumed 40
Produced 55
Consumed 55
Produced 7
Consumed 7
Produced 84
Consumed 84
Produced 84
Consumed 84
```

Q. The Dining Philosopher Problem – The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

There are three states of philosopher : THINKING, HUNGRY and EATING. Here there are two semaphores : Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

Solution:

```
import threading
import random
import time

#inheriting threading class in Thread module
class Philosopher_331(threading.Thread):
    running = True  #used to check if everyone is finished eating

 #Since the subclass overrides the constructor, it must make sure to invoke the base class
constructor
    #(Thread.__init__()) before doing anything else to the thread.
    def __init__(self, index, forkOnLeft, forkOnRight):
        threading.Thread.__init__(self)
        self.index = index
        self.forkOnLeft = forkOnLeft
        self.forkOnRight = forkOnRight

    def run(self):
        while(self.running):
            # Philosopher is thinking (but really is sleeping).
            time.sleep(30)
            print ('Philosopher %s is hungry.' % self.index)
            self.dine_331()

    def dine_331(self):
```

```
        # if both the semaphores(forks) are free, then philosopher will eat
        fork1, fork2 = self.forkOnLeft, self.forkOnRight
        while self.running:
            fork1.acquire() # wait operation on left fork
            locked = fork2.acquire(False)
            if locked: break #if right fork is not available leave left fork
            fork1.release()
            print ('Philosopher %s swaps forks.' % self.index)
            fork1, fork2 = fork2, fork1
        else:
            return
        self.dining_331()
        #release both the fork after dining
        fork2.release()
        fork1.release()

    def dining_331(self):
        print ('Philosopher %s starts eating. '% self.index)
        time.sleep(30)
        print ('Philosopher %s finishes eating and leaves to think.' % self.index)

def main():
    forks = [threading.Semaphore() for n in range(5)] #initialising array of semaphore i.e forks

    #here (i+1)%5 is used to get right and left forks circularly between 1-5
    philosophers= [Philosopher_331(i, forks[i%5], forks[(i+1)%5])
            for i in range(5)]

    Philosopher_331.running = True
    for p in philosophers: p.start()
    time.sleep(100)
    Philosopher_331.running = False
    print ("Now we're finishing.")


if __name__ == "__main__":
    main()
```

Output:

```
if __name__ == "__main__":
    main()
```

```
Philosopher 0 is hungry.
Philosopher 0 starts eating.
Philosopher 4 is hungry.
Philosopher 4 swaps forks.
Philosopher 2 is hungry.
Philosopher 2 starts eating.
Philosopher 1 is hungry.
Philosopher 3 is hungry.
Philosopher 0 finishes eating and leaves to think.
Philosopher 1 swaps forks.
Philosopher 4 starts eating.
Philosopher 2 finishes eating and leaves to think.
Philosopher 1 starts eating. Philosopher 3 swaps forks.

Philosopher 4 finishes eating and leaves to think.Philosopher 0 is hungry.

Philosopher 0 swaps forks.Philosopher 3 starts eating.

Philosopher 1 finishes eating and leaves to think.Philosopher 2 is hungry.

Philosopher 0 starts eating. Philosopher 2 swaps forks.

Now we're finishing.
Philosopher 3 finishes eating and leaves to think.Philosopher 4 is hungry.

Philosopher 2 starts eating.
Philosopher 0 finishes eating and leaves to think.Philosopher 1 is hungry.

Philosopher 2 finishes eating and leaves to think.
```