**THREADS - 2**

**THREAD SYNCHRONIZATION**

When multiple threads are accessing the same resources it is very important that they do not corrupt each other resources or objects.

In Thread Synchronization, we can lock a particular thread using two different ways, using locks or semaphores.

LOCK:

When you lock an object, it enters into a room of its own. It will take that object and it will own that object and only when the thread releases the object the other threads can use that object or resources.

The process of thread acquiring a lock and entering a room is also known as thread mutex.

A blue rectangle with black text

Description automatically generated

To acquire a lock on an object, first we have to create a lock object and invoke it using lock.acquire() method.

A black text with black text

Description automatically generated with medium confidence

Once it acquires the lock, until it invokes the release method no other thread can process or use

A black text on a white background

Description automatically generated

the current object.

SEMAPHORE:

Similarly, we can use semaphore. Semaphore is simply acquiring a lock, but internally it uses a counter.

Internally, initially there will be a number 1 and every time a lock is acquired this number will be decremented by 1 which will be zero.

Once the lock is released, it will be incremented by 1 again.

The process is the same as lock :

A white background with black text

Description automatically generated

**LOCK IN MULTI TREADING**

**Code**

from threading import \*

class **BookTickets**:

def **\_\_init\_\_**(*self*,sAvailable):

*self*.availableSeats = sAvailable

*self*.locking=Lock()

def **buyTickets**(*self*,reqSeats):

# LOCK THIS SET OF EXECUTION USING ACQUIRE

*self*.locking.acquire()

print(*"TOTAL SEATS AVAILABLE : "*, *self*.availableSeats)

if(*self*.availableSeats >= reqSeats):

print(*"REQUESTED SEATS : "*,reqSeats)

print(*"Required Seats AVAILABLE ......"*)

print(*"Selecting seat to confirm....."*)

print(*"Processing the payment for the selected seats..."*)

print(*"Seats Confirmed.... Emailing the confirmation !!!"*)

*self*.availableSeats-=reqSeats

print()

else:

print(*"REQUESTED SEATS : "*,reqSeats)

print(*"Required Seats NOT AVAILABLE !!!!!"*)

print()

# RELEASE THE LOCK USING RELEASE

*self*.locking.release()

#CREATING OBJECT OF THE CLASS

bt = BookTickets(20)

#WHEN PASSING VALUES FOR args PASS IN AS ITERATOR OR LIST

# THREAD WILL REQUIED AS ITERATOR OR LIST, HENCE GIVE THE VALUE AS (VALUE,)

thread1 = Thread(target=bt.buyTickets, args=(5,))

thread2 = Thread(target=bt.buyTickets, args=(12,))

thread3 = Thread(target=bt.buyTickets, args=(10,))

thread1.start()

thread2.start()

thread3.start()

**Output:**

TOTAL SEATS AVAILABLE : 20

REQUESTED SEATS : 5

Required Seats AVAILABLE ......

Selecting seat to confirm.....

Processing the payment for the selected seats...

Seats Confirmed.... Emailing the confirmation !!!

TOTAL SEATS AVAILABLE : 15

REQUESTED SEATS : 12

Required Seats AVAILABLE ......

Selecting seat to confirm.....

Processing the payment for the selected seats...

Seats Confirmed.... Emailing the confirmation !!!

TOTAL SEATS AVAILABLE : 3

REQUESTED SEATS : 10

Required Seats NOT AVAILABLE !!!!!

**SEMAPHORE IN MULTI TREADING**

**Code**

from threading import \*

class **BookTickets**:

def **\_\_init\_\_**(*self*,sAvailable):

*self*.availableSeats = sAvailable

#self.locking=Lock()

*self*.locking=Semaphore()

def **buyTickets**(*self*,reqSeats):

# LOCK THIS SET OF EXECUTION USING ACQUIRE

*self*.locking.acquire()

print(*"TOTAL SEATS AVAILABLE : "*, *self*.availableSeats)

if(*self*.availableSeats >= reqSeats):

print(*"REQUESTED SEATS : "*,reqSeats)

print(*"Required Seats AVAILABLE ......"*)

print(*"Selecting seat to confirm....."*)

print(*"Processing the payment for the selected seats..."*)

print(*"Seats Confirmed.... Emailing the confirmation !!!"*)

*self*.availableSeats-=reqSeats

print()

else:

print(*"REQUESTED SEATS : "*,reqSeats)

print(*"Required Seats NOT AVAILABLE !!!!!"*)

print()

# RELEASE THE LOCK USING RELEASE

*self*.locking.release()

#CREATING OBJECT OF THE CLASS

bt = BookTickets(20)

#WHEN PASSING VALUES FOR args PASS IN AS ITERATOR OR LIST

# THREAD WILL REQUIED AS ITERATOR OR LIST, HENCE GIVE THE VALUE AS (VALUE,)

thread1 = Thread(target=bt.buyTickets, args=(5,))

thread2 = Thread(target=bt.buyTickets, args=(12,))

thread3 = Thread(target=bt.buyTickets, args=(10,))

thread1.start()

thread2.start()

thread3.start()

**Output:**

SAME OUTPUT AS ABOVE

**THREAD COMMUNICATION**

When we work on real time applications which use multiple threads, often these threads need to communicate with each other to get a job done.

A very common pattern we see in muli-threaded application is the producer and consumer pattern where we have two threads, producer thread and consumer thread.

The producer thread is responsible for creating some orders.

The Consumer thread is responsible for processing the orders and shipping them.

These 2 threads have to communicate with each other when the list of orders are available for processing.

A diagram of a communication diagram

Description automatically generated

The Consumer thread will be communicating with the Producer thread and checking for the value orderplaced = true.

One way of using the communication is by setting a Boolean flag.

**Code:**

from threading import \*

from time import \*

class **Producer**:

def **\_\_init\_\_**(*self*):

*self*.orderedProducts=[]

*self*.orderPlaced= False

def **produce**(*self*):

for i in range(10,15):

product = (*"iPhone"*+str(i))

*self*.orderedProducts.append(product)

sleep(2)

print(product,*" added in Cart...."*)

*self*.orderPlaced=True

print()

class **Consumer**:

def **\_\_init\_\_**(*self*, prodThread):

*self*.prodThread = prodThread

def **consume**(*self*):

while *self*.prodThread.orderPlaced == False:

sleep(0.2)

print(*"Consumer Thread Running....."*)

print (*"Orders Shipped : "*, *self*.prodThread.orderedProducts)

prod = Producer()

con = Consumer(prod)

thread1 = Thread(target=prod.produce)

thread2 = Thread(target=con.consume)

print(*"Producer Thread Running....."*)

thread1.start()

thread2.start()

**Output:**

Producer Thread Running.....

iPhone10 added in Cart....

iPhone11 added in Cart....

iPhone12 added in Cart....

iPhone13 added in Cart....

iPhone14 added in Cart....

Consumer Thread Running.....

Orders Shipped : ['iPhone10', 'iPhone11', 'iPhone12', 'iPhone13', 'iPhone14']

**THREAD COMMUNICATION USING API METHODS**

The second and most powerful way for threads to communicate to use the multi-threading API methods such as wait, notify and notifyAll.

These methods are available on a class called Condition from the multi-threading API in Python.

Once we use the Condition class in the producer, we can access this Condition, that condition object in the consumer and invoke the wait method.

When the Producer method invoke the notify method, whenever its ready with its list.

If multiple methods are waiting for the task to be done, then we will invoke the notifyAll method.

All these process should happen in a lock context.

A diagram of a computer connection

Description automatically generated

**Code:**

from threading import \*

from time import \*

class **Producer**:

def **\_\_init\_\_**(*self*):

*self*.orderedProducts=[]

# ADD A CONDTION CLASS AVAIALBLE FROM THREADING MODULE WHICH WE

# \_ARE GOING TO USE TO COMMUNICATE BETWEEN THREAD CONDITIONALLY

*self*.cond = Condition()

def **produce**(*self*):

# ACQUIRE A LOCK ON THE cond (CONDITION OBJECT)

*self*.cond.acquire()

for i in range(10,15):

product = (*"Samsung S"*+str(i))

*self*.orderedProducts.append(product)

sleep(2)

print(product,*" added in Cart...."*)

#WHEN THE ABOVE JOB IS DONE

*self*.cond.notify()

#NOW RELEASE THE LOCK

*self*.cond.release()

print()

class **Consumer**:

def **\_\_init\_\_**(*self*, prodThread):

*self*.prodThread = prodThread

def **consume**(*self*):

# TO DO IN A SYNCHRONIZED CONTEXT,

*self*.prodThread.cond.acquire()

# INVOKE THE WAIT METHOD

*self*.prodThread.cond.wait(timeout=0)

print(*"Consumer Thread Running....."*)

print (*"Orders Shipped : "*, *self*.prodThread.orderedProducts)

#NOW RELEASE THE LOCK

*self*.prodThread.cond.release()

prod = Producer()

con = Consumer(prod)

thread1 = Thread(target=prod.produce)

thread2 = Thread(target=con.consume)

print(*"Producer Thread Running....."*)

thread1.start()

thread2.start()

**Output:**

Producer Thread Running.....

Samsung S10 added in Cart....

Samsung S11 added in Cart....

Samsung S12 added in Cart....

Samsung S13 added in Cart....

Samsung S14 added in Cart....

Consumer Thread Running.....

Orders Shipped : ['Samsung S10', 'Samsung S11', 'Samsung S12', 'Samsung S13', 'Samsung S14']

**INTERTHREAD COMMUNICATION USING QUEUE**

Queue (Q) is an inbuild class or object in Python which us the methods that make it super easy to do Inter thread communication.

The Queue object has two methods :

put : puts the item onto the queue.

This will lock the queue object when it performs the put operation, so that get will not happen. No other thread will be able to access the queue.

get : gets the item from queue by removing it from the queue.

This method will also lock the queue. So the synchronization is taken care of by the logic which is already there in the put and get methods of the queue internally.

Both have some conditional locking happening.

A diagram of a diagram

Description automatically generated

**Code:**

import random  
import time  
import queue  
from threading import \*  
  
  
def producer(que):  
 while True:  
 print("Producing..........")  
 que.put(random.randint(145,150))  
 print("Produced")  
 time.sleep(3)  
  
def consumer(que):  
 while True:  
 print("Consuming......")  
 print("Consumer Data : ", que.get())  
 time.sleep(3)  
  
  
que = queue.Queue()  
thread1 = Thread(target=consumer, args=(que,))  
thread2 = Thread(target=producer, args=(que,))  
  
  
thread1.start()  
thread2.start()

**Output: <OUTPUT IS GOING ON INFINITY…>**

C:\Users\kamal\pythonlab\Scripts\python.exe C:\Users\kamal\PycharmProjects\pythonProjectdemo\producerConsumer.py

ProducedConsumer Data : 150

Producing..........

Produced

Consuming......

Consumer Data : 146

Producing..........Consuming......

Produced

Consumer Data : 150

Producing..........

Produced

Consuming......

Consumer Data : 145

Producing..........

Produced

Consuming......

Process finished with exit code -1

**TYPES OF QUEUE EXAMPLE**

**Code :**

import queue  
  
*# FIFO METHOD*print("FIFI Queue Output")  
print("-------------------------")  
que1 = queue.Queue()  
que1.put("Augusta")  
que1.put("Jerrick")  
que1.put("Jazlyn")  
que1.put("Kumar")  
  
while not que1.empty():  
 print(que1.get(), end=' ')  
  
print("\n")  
  
*# LIFO METHOD*print("LIFO Queue Output")  
print("-------------------------")  
que2 = queue.LifoQueue()  
que2.put("Augusta")  
que2.put("Jerrick")  
que2.put("Jazlyn")  
que2.put("Kumar")  
  
while not que2.empty():  
 print(que2.get(), end=' ')  
  
print("\n")  
  
*# PRIORITY METHOD*print("PRIORITY Queue Output")  
print("-------------------------")  
que3 = queue.PriorityQueue()  
que3.put("Augusta")  
que3.put("Jerrick")  
que3.put("Jazlyn")  
que3.put("Kumar")  
  
while not que3.empty():  
 print(que3.get(), end=' ')

**Output:**

C:\Users\kamal\pythonlab\Scripts\python.exe C:\Users\kamal\PycharmProjects\pythonProjectdemo\queueTypes.py

FIFI Queue Output

-------------------------

Augusta Jerrick Jazlyn Kumar

LIFO Queue Output

-------------------------

Kumar Jazlyn Jerrick Augusta

PRIORITY Queue Output

-------------------------

Augusta Jazlyn Jerrick Kumar

Process finished with exit code 0

**TYPES OF QUEUE EXAMPLE (USING TUPLE)**

**Code :**

import queue  
  
*# FIFO METHOD USING TUPLE*print("FIFI Queue Output")  
print("-------------------------")  
que1 = queue.Queue()  
que1.put((5,"Augusta"))  
que1.put((3,"Jerrick"))  
que1.put((4,"Jazlyn"))  
que1.put((2,"Kumar"))  
  
while not que1.empty():  
 print(que1.get(), end=' ')  
 '''print("Get one value from Tuple")  
 print("-------------------------")  
 print(que1.get()[1],end=' ')'''  
  
print("\n")  
  
*# LIFO METHOD USING TUPLE*print("LIFO Queue Output")  
print("-------------------------")  
que2 = queue.LifoQueue()  
que2.put((5,"Augusta"))  
que2.put((3,"Jerrick"))  
que2.put((4,"Jazlyn"))  
que2.put((2,"Kumar"))  
  
while not que2.empty():  
 print(que2.get(), end=' ')  
 '''print("Get one value from Tuple")  
 print("-------------------------")  
 print(que2.get()[1], end=' ')'''  
  
print("\n")  
  
*# PRIORITY METHOD USING TUPLE*print("PRIORITY Queue Output")  
print("-------------------------")  
que3 = queue.PriorityQueue()  
que3.put((5,"Augusta"))  
que3.put((3,"Jerrick"))  
que3.put((4,"Jazlyn"))  
que3.put((2,"Kumar"))  
  
while not que3.empty():  
 print(que3.get(), end=' ')  
 '''print("-------------------------")  
 print("Get one value from Tuple")  
 print(que3.get()[1],end=' ')'''

**Output:**

C:\Users\kamal\pythonlab\Scripts\python.exe C:\Users\kamal\PycharmProjects\pythonProjectdemo\queueTypes.py

FIFI Queue Output

-------------------------

(5, 'Augusta') (3, 'Jerrick') (4, 'Jazlyn') (2, 'Kumar')

LIFO Queue Output

-------------------------

(2, 'Kumar') (4, 'Jazlyn') (3, 'Jerrick') (5, 'Augusta')

PRIORITY Queue Output

-------------------------

(2, 'Kumar') (3, 'Jerrick') (4, 'Jazlyn') (5, 'Augusta')

Process finished with exit code 0

**TYPES OF QUEUE EXAMPLE (USING TUPLE / INDEX)**

**Code :**

import queue  
  
*# FIFO METHOD USING TUPLE*print("FIFI Queue Output - USING TYPLE INDEX")  
print("-------------------------------------")  
que1 = queue.Queue()  
que1.put((5,"Augusta"))  
que1.put((3,"Jerrick"))  
que1.put((4,"Jazlyn"))  
que1.put((2,"Kumar"))  
  
while not que1.empty():  
 *#print(que1.get(), end=' ')* print(que1.get()[1],end=' ')  
  
print("\n")  
  
*# LIFO METHOD USING TUPLE*print("LiFO Queue Output - USING TYPLE INDEX")  
print("-------------------------------------")  
que2 = queue.LifoQueue()  
que2.put((5,"Augusta"))  
que2.put((3,"Jerrick"))  
que2.put((4,"Jazlyn"))  
que2.put((2,"Kumar"))  
  
while not que2.empty():  
 *#print(que2.get(), end=' ')* print(que2.get()[1], end=' ')  
  
print("\n")  
  
*# PRIORITY METHOD USING TUPLE*print("PRIORITY Queue Output - USING TYPLE INDEX")  
print("-----------------------------------------")  
que3 = queue.PriorityQueue()  
que3.put((5,"Augusta"))  
que3.put((3,"Jerrick"))  
que3.put((4,"Jazlyn"))  
que3.put((2,"Kumar"))  
  
while not que3.empty():  
 *#print(que3.get(), end=' ')* print(que3.get()[1],end=' ')

**Output:**

C:\Users\kamal\pythonlab\Scripts\python.exe C:\Users\kamal\PycharmProjects\pythonProjectdemo\queueTypes.py

FIFI Queue Output - USING TYPLE INDEX

-------------------------------------

Augusta Jerrick Jazlyn Kumar

LiFO Queue Output - USING TYPLE INDEX

-------------------------------------

Kumar Jazlyn Jerrick Augusta

PRIORITY Queue Output - USING TYPLE INDEX

-----------------------------------------

Kumar Jerrick Jazlyn Augusta

Process finished with exit code 0

**QUIZ**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

ASSIGNMENT CODES

**Code 1:**

import threading  
from threading import \*  
from time import sleep  
  
class EvenThread:  
 def processEven(self):  
 print(threading.current\_thread().name)  
 for i in range(1,101):  
 if(i%2 ==0):  
 print(i)  
  
class OddThread:  
 def processOdd(self):  
 print(threading.current\_thread().name)  
 for i in range(1,101):  
 if(i%2 != 0):  
 print(i)  
  
print("Even Numbers from 1 to 100")  
ev = EvenThread()  
thEven = Thread(target=ev.processEven)  
thEven.start()  
print("\n")  
  
sleep(10)  
  
print("Even Numbers from 1 to 100")  
od = OddThread()  
thOdd = Thread(target=od.processOdd())  
thOdd.start()  
print("\n")  
  
sleep(10)  
  
print("All the numbers from 1 to 100")  
for i in range(1,101):  
 print(i)

**Code 2:**

from threading import \*

def odd():

    lock.acquire()

    print(current\_thread().getName())

    for i in range(1,10,2): print(i)

    lock.release()

def even():

    lock.acquire()

    print(current\_thread().getName())

    for i in range(2,11,2): print(i)

    lock.release()

lock = Lock()

even\_thread = Thread(target=even)

odd\_thread = Thread(target=odd)

even\_thread.start()

odd\_thread.start()

lock.acquire()

print(current\_thread().getName())

for i in range(1,11): print(i)

lock.release()

**Code 3:**

from threading import \*

class Print100:

    def even100(self):

    ##Even

        for i in range(1,101):

            if i%2==0:

                print(i)

    def odd100(self):

    ##Odd

        for i in range(1,101):

            if i%2!=0:

                print(i)

    def all100(self):

    ##All

        for i in range(1,101):

            print(i)

obj=Print100()

t1=Thread(target=obj.all100)

t2=Thread(target=obj.even100)

t3=Thread(target=obj.odd100)

t1.start()

t2.start()

t3.start()