APP WEEK-8 LAB

Q1.
Implement a stack as a linked list in which the push, pop, and isEmpty methods can be safely accessed from multiple threads

Code:

```
import threading
class Node:
    def _init_(self, data):
        self.data = data
        self.next = None
class TheadSafeStack:
    def _init_(self):
        self.lock = threading.Lock()
        self.head = None
    def push(self, data):
        with self.lock:
            new_node = Node(data)
            new_node.next = self.head
            self.head = new_node
    def pop(self):
        with self.lock:
            if self.head is None:
                return None
            else:
                popped_node = self.head
                self.head = self.head.next
                popped_node.next = None
                return popped_node.data
    def isEmpty(self):
        with self.lock:
            return self.head is None
if _name_ == "_main_":
    stack = ThreadSafeStack()
    # Create and start some threads to push and pop from the stack
    def push_thread():
        for i in range(5):
            stack.push(i)
    def pop_thread():
        for i in range(5):
```

```
        value = stack.pop()
        print(f"Popped value: {value}")
t1 = threading.Thread(target=push_thread)
t2 = threading.Thread(target=pop_thread)
t1.start()
t2.start()
t1.join()
t2.join()
# Check if the stack is empty
print(f"Is stack empty? {stack.isEmpty()}")
```

SnapShot:

```
Popped value: 4
Popped value: 3
Popped value: 2
Popped value: 1
Popped value: 0
Is stack empty? True
```

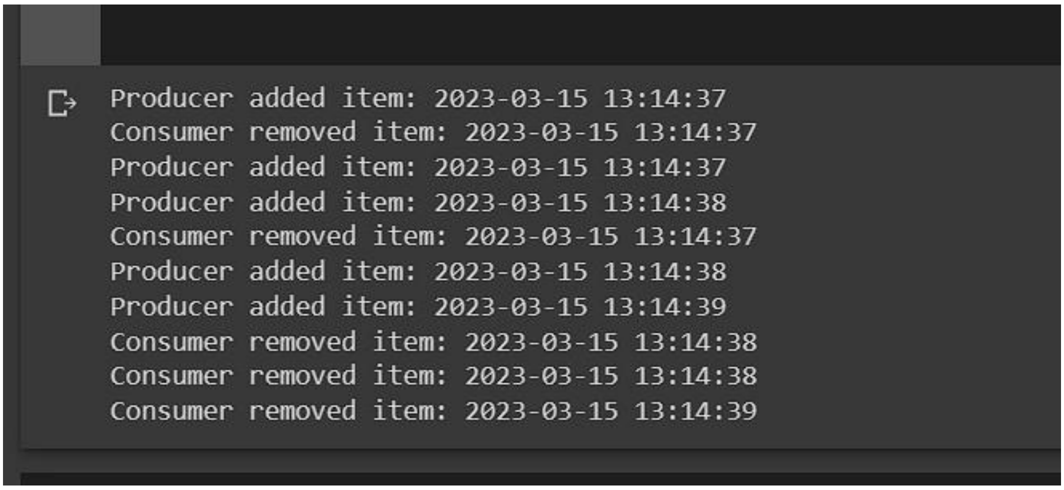
Q2.
Implement a Queue class whose add and remove methods are synchronized. Supply one thread, called the producer, which keeps inserting strings into the queue as long as there are fewer than ten elements in it. When the queue gets too full, the thread waits. As sample strings, simply use time stamps new Date().toString(). Supply a sec ond thread, called the consumer, that keeps removing and printing strings from the queue as long as the queue is not empty. When the queue is empty, the thread waits. Both the consumer and producer threads should run for 3 iterations.

Code:

```
import threading
import time

class Queue:
    def_init_(self):
        self.items = []
        self.lock = threading.Lock()

    def add(self, item):
        with self.lock:
            self.items.append(item)

    def remove(self):
        with self.lock:
            if self.items:
                return self.items.pop(0)
            else:
                return None

def producer(q):
    for i in range(5):
        while len(q.items) >= 5:
            time.sleep(1)
        item = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
        q.add(item)
        print(f"Producer added item: {item}")
        time.sleep(0.5)

def consumer(q):
    for i in range(5):
        item = q.remove()
        while not item:
            time.sleep(1)
            item = q.remove()
        print(f"Consumer removed item: {item}")
        time.sleep(1)
```

```
q = Queue()
t1 = threading.Thread(target=producer, args=(q,))
t2 = threading.Thread(target=consumer, args=(q,))
t1.start()
t2.start()
t1.join()
t2.join()
```

<u>SnapShot:</u>



```
Producer added item: 2023-03-15 13:14:37
Consumer removed item: 2023-03-15 13:14:37
Producer added item: 2023-03-15 13:14:37
Producer added item: 2023-03-15 13:14:38
Consumer removed item: 2023-03-15 13:14:37
Producer added item: 2023-03-15 13:14:38
Producer added item: 2023-03-15 13:14:39
Consumer removed item: 2023-03-15 13:14:38
Consumer removed item: 2023-03-15 13:14:38
Consumer removed item: 2023-03-15 13:14:39
```

Q3.

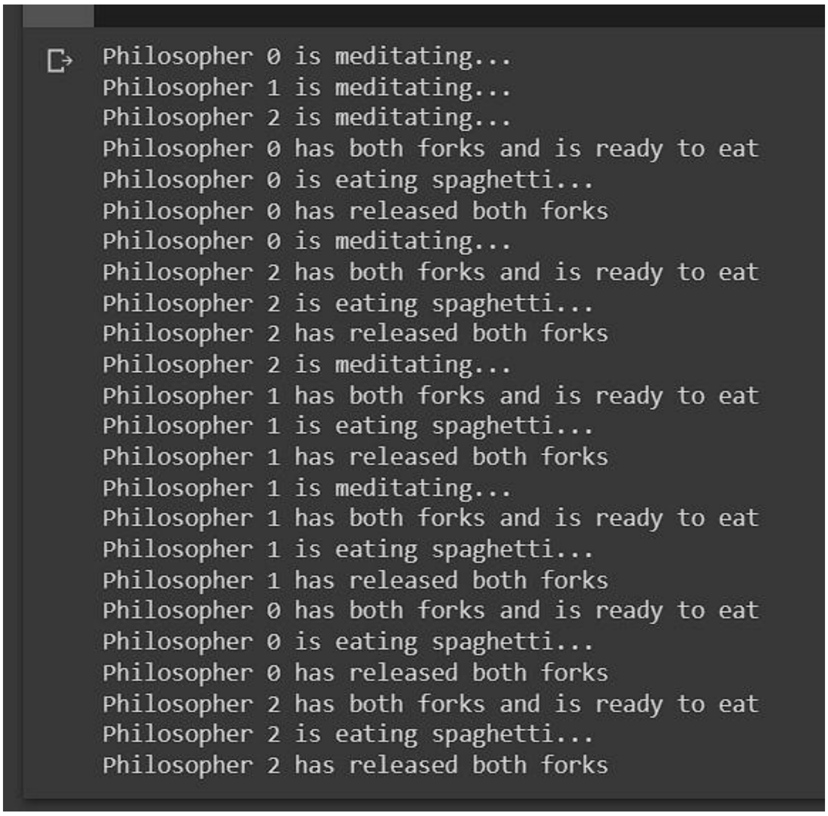
N philosophers sit at a table with a plate of spaghetti in front of them and a fork on their right and one on their left. To eat spaghetti, a philosopher needs both forks close together. Each philosopher is continuously engaged in a sequence of 2 activities: meditating, trying to acquire forks and eating. Write a program that activates N philosopher threads that execute the described loop 3 times. Meditation and the phase where the philosopher eats must be implemented with a variable delay (use for example the sleep call and the rand() function)

<u>Code:</u>

```
import threading
import time
import random
class Philosopher(threading.Thread):
    def_init_(self, name, left_fork, right_fork):
        super()._init_(name=name)
        self.left_fork =  left_fork
        self.right_fork = right_fork
    def run(self):
        for i in range(2):
            self.meditate()
            self.acquire_forks()
            self.eat()
            self.release_forks()
    def meditate(self):
        print(f"{self.name} is meditating...")
        time.sleep(random.uniform(0, 1))
```

```
    def acquire_forks(self):
        while True:
            if self.left_fork.acquire(blocking=False):
                if self.right_fork.acquire(blocking=False):
                    print(f"{self.name} has both forks and is ready to eat")
                    break
                else:
                    self.left_fork.release()
            time.sleep(random.uniform(0, 1))
    def eat(self):
        print(f"{self.name} is eating spaghetti...")
        time.sleep(random.uniform(0, 1))
    def release_forks(self):
        self.left_fork.release()
        self.right_fork.release()
        print(f"{self.name} has released both forks")
if__name__== '_main_':
    N = 3
    forks = [threading.Lock() for i in range(N)]
    philosophers = [Philosopher(f"Philosopher {i}", forks[i], forks[(i+1)%N]) for i in range(N)]
    for p in philosophers:
        p.start()
    for p in philosophers:
        p.join()
```

SnapShot:



```
Philosopher 0 is meditating...
Philosopher 1 is meditating...
Philosopher 2 is meditating...
Philosopher 0 has both forks and is ready to eat
Philosopher 0 is eating spaghetti...
Philosopher 0 has released both forks
Philosopher 0 is meditating...
Philosopher 2 has both forks and is ready to eat
Philosopher 2 is eating spaghetti...
Philosopher 2 has released both forks
Philosopher 2 is meditating...
Philosopher 1 has both forks and is ready to eat
Philosopher 1 is eating spaghetti...
Philosopher 1 has released both forks
Philosopher 1 is meditating...
Philosopher 1 has both forks and is ready to eat
Philosopher 1 is eating spaghetti...
Philosopher 1 has released both forks
Philosopher 0 has both forks and is ready to eat
Philosopher 0 is eating spaghetti...
Philosopher 0 has released both forks
Philosopher 2 has both forks and is ready to eat
Philosopher 2 is eating spaghetti...
Philosopher 2 has released both forks
```

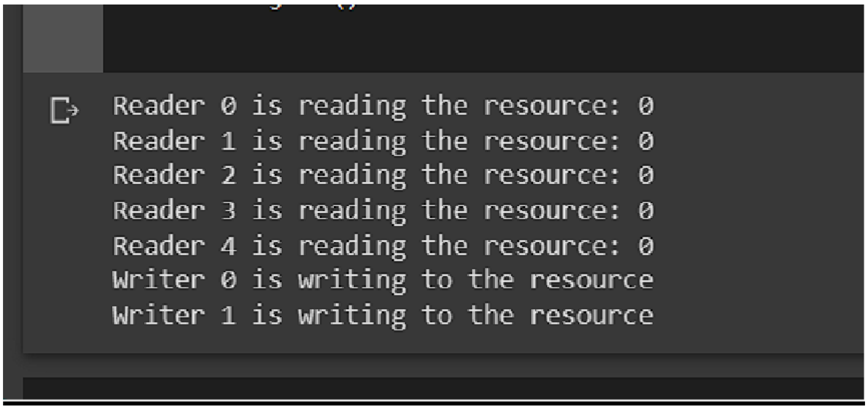
Q4.
Reader-Writer Problem: This is a classic problem that demonstrates the use of synchronization in Java. The goal is to have multiple readers reading a shared resource. simultaneously, while a writer can modify the resource. The challenge is to ensure that readers do not interfere with each other and that the writer has exclusive access to the resource when making modifications.

<u>Code:</u>

```
import threading
import time
import random

class SharedResource:
    def_init_(self):
        self.resource = 0
        self.reader_count = 0
        self.writer_count = 0
        self.lock = threading.Lock()
        self.read_cv = threading.Condition(self.lock)
        self.write_cv = threading.Condition(self.lock)

    def read(self, reader_id):
        with self.read_cv:
            while self.writer_count > 0:
                self.read_cv.wait()
            self.reader_count += 1
            print(f"Reader {reader_id} is reading the resource: {self.resource}")
            time.sleep(random.uniform(0, 1))
            self.reader_count -= 1
            if self.reader_count == 0:
                self.write_cv.notify()

    def write(self, writer_id):
        with self.write_cv:
            while self.writer_count > 0 or self.reader_count > 0:
                self.write_cv.wait()
            self.writer_count += 1
            print(f"Writer {writer_id} is writing to the resource")
            self.resource += 1
            time.sleep(random.uniform(0, 1))
            self.writer_count -= 1
            self.read_cv.notify_all()
            self.write_cv.notify()

if__name__== '_main_':
    resource = SharedResource()
    N_READERS = 5
    N_WRITERS = 2

    readers = [threading.Thread(target=resource.read, args=(i,)) for i in range(N_READERS)]
    writers = [threading.Thread(target=resource.write, args=(i,)) for i in range(N_WRITERS)]
```

```
  for r in readers:
      r.start()

  for w in writers:
      w.start()

  for r in readers:
      r.join()

  for w in writers:
      w.join()
```

SnapShot:



Q5.
Sleeping Barber Problem: This problem is used to demonstrate the use of synchronization and inter-thread communication in Python. The goal is to model the behavior of a barber shop where customers arrive to get haircuts and the barber is responsible for cutting their hair. The challenge is to ensure that customers are served in the order in which they arrive, and that the barber does not start cutting hair until a customer is available.

Code:

```
import threading
import time
import random

MAX_CUSTOMERS = 2
waiting_room = []
barber_sleeping = threading.Event()

class Customer:
    def_init_(self, id):
        self.id = id

def barber():
    while True:
        print("Barber falls asleep")
        barber_sleeping.wait()
        while len(waiting_room) > 0:
            customer = waiting_room[0]
```

```
            waiting_room.remove(customer)
            print(f"Barber is cutting hair of customer {customer.id}")
            time.sleep(random.randint(1, 3))
            print(f"Customer {customer.id} leaves the barber shop")
        else:
            print("No more customers in the queue, barber goes back to sleep")
            barber_sleeping.clear()

def customer_arrives():
    id = 0
    while id < MAX_CUSTOMERS:
        time.sleep(random.randint(1, 4))
        customer = Customer(id)
        print(f"Customer {customer.id} arrives")
        if len(waiting_room) < MAX_CUSTOMERS:
            waiting_room.append(customer)
            print(f"Customer {customer.id} takes a seat in the waiting room")
            if barber_sleeping.is_set():
                barber_sleeping.clear()
                print("Barber wakes up")
        else:
            print(f"Waiting room is full, customer {customer.id} leaves")
            id -= 1
        id += 1
    print("All customers have arrived, shop is closing")
    barber_sleeping.set()

barber_thread = threading.Thread(target=barber, daemon=True)
customer_thread = threading.Thread(target=customer_arrives, daemon=True)

barber_thread.start()
customer_thread.start()

time.sleep(15)
```

<u>SnapShot:</u>

```
Barber falls asleep
Customer 0 arrives
Customer 0 takes a seat in the waiting room
Barber wakes up
Barber is cutting hair of customer 0
Customer 0 leaves the barber shop
Customer 1 arrives
Customer 1 takes a seat in the waiting room
Barber is cutting hair of customer 1
Customer 1 leaves the barber shop
All customers have arrived, shop is closing
No more customers in the queue, barber goes back to sleep
```

Q6.
Write a python program to Print alternate numbers using 2 Threads. Implement using wait and notify construct.

Code:

```
import threading

class NumberPrinter:
    def_init_(self, max_number):
        self.max_number = max_number
        self.current_number = 1
        self.lock = threading.Lock()
        self.cond_var = threading.Condition(self.lock)

    def print_numbers(self):
        while self.current_number <= self.max_number:
            with self.lock:
                print(threading.current_thread().name, self.current_number)
                self.current_number += 1
                self.cond_var.notify()

                if self.current_number > self.max_number:
                    return

                self.cond_var.wait()

def thread_one(printer):
    printer.print_numbers()

def thread_two(printer):
    printer.print_numbers()

if__name__== '_main_':
    printer =NumberPrinter(20)

    t1 = threading.Thread(target=thread_one, args=(printer,))
    t2 = threading.Thread(target=thread_two, args=(printer,))

    t1.start()
    t2.start()

    t1.join()
    t2.join()
```

SnapShot:

```
Thread-23 1
Thread-24 2
Thread-23 3
Thread-24 4
Thread-23 5
Thread-24 6
Thread-23 7
Thread-24 8
Thread-23 9
Thread-24 10
Thread-23 11
Thread-24 12
Thread-23 13
Thread-24 14
Thread-23 15
Thread-24 16
Thread-23 17
Thread-24 18
Thread-23 19
Thread-24 20
```

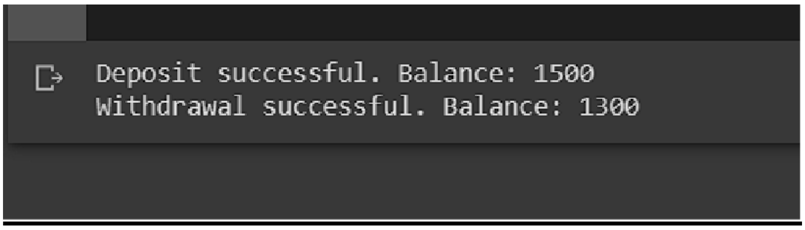
Q7.
Write a python program to implement banking account with necessary function.
Ensure both withdrawal and deposit can be carried out safely by employing concurrency control.

Code:

```
import threading

class BankAccount:
    def_init_(self, balance=0):
        self.balance = balance
        self.lock = threading.Lock()

    def deposit(self, amount):
        with self.lock:
            self.balance += amount
            print(f'Deposit successful. Balance: {self.balance}')

    def withdraw(self, amount):
        with self.lock:
            if self.balance >= amount:
                self.balance -= amount
                print(f'Withdrawal successful. Balance: {self.balance}')
                return True
            else:
                print('Insufficient balance.')
                return False
```

```
def deposit(account, amount):
    account.deposit(amount)

def withdraw(account, amount):
    account.withdraw(amount)

if__name__== '_main_':
    account = BankAccount(1000)

    t1 = threading.Thread(target=deposit, args=(account, 500))
    t2 = threading.Thread(target=withdraw, args=(account, 200))

    t1.start()
    t2.start()

    t1.join()
    t2.join()
```

SnapShot:



```
Deposit successful. Balance: 1500
Withdrawal successful. Balance: 1300
```