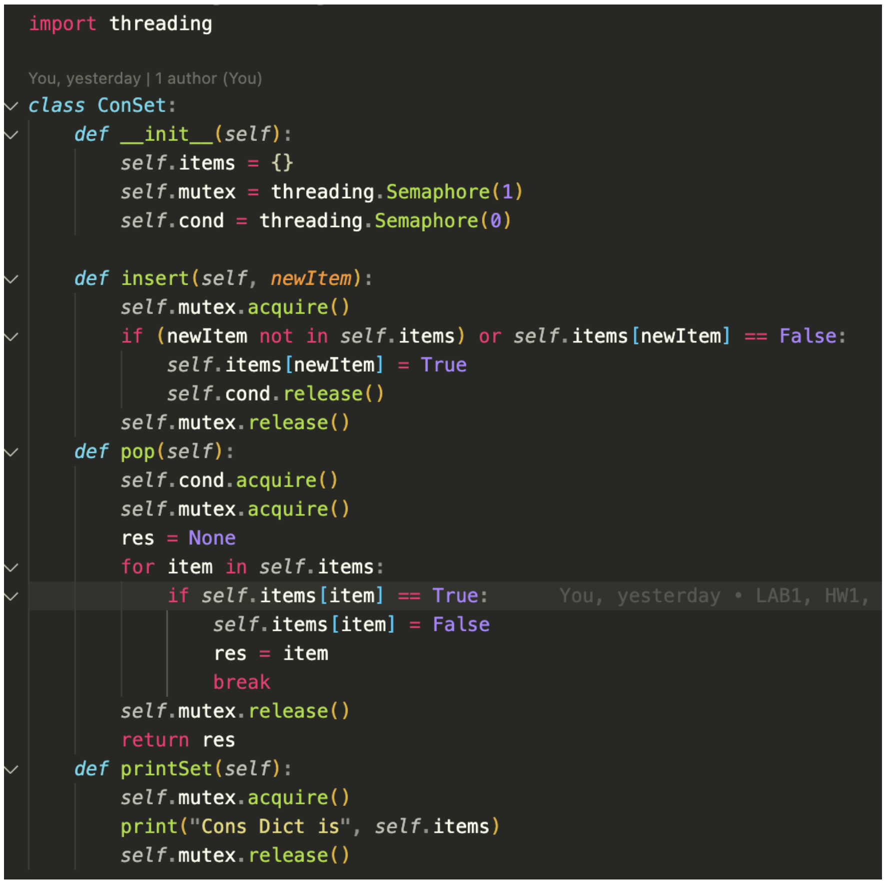
CS 403 Assignment 1 Report

Gorkem Yar
27970

For this assignment, we expected to write to Python files named ConSet.py and Leader.py. The ConSet.py file includes an implementation of a set class that allows concurrent insertions and pop operations. It guarantees linearizability by using Semaphores as locks and condition variables.

This is the implementation of the ConSet class.

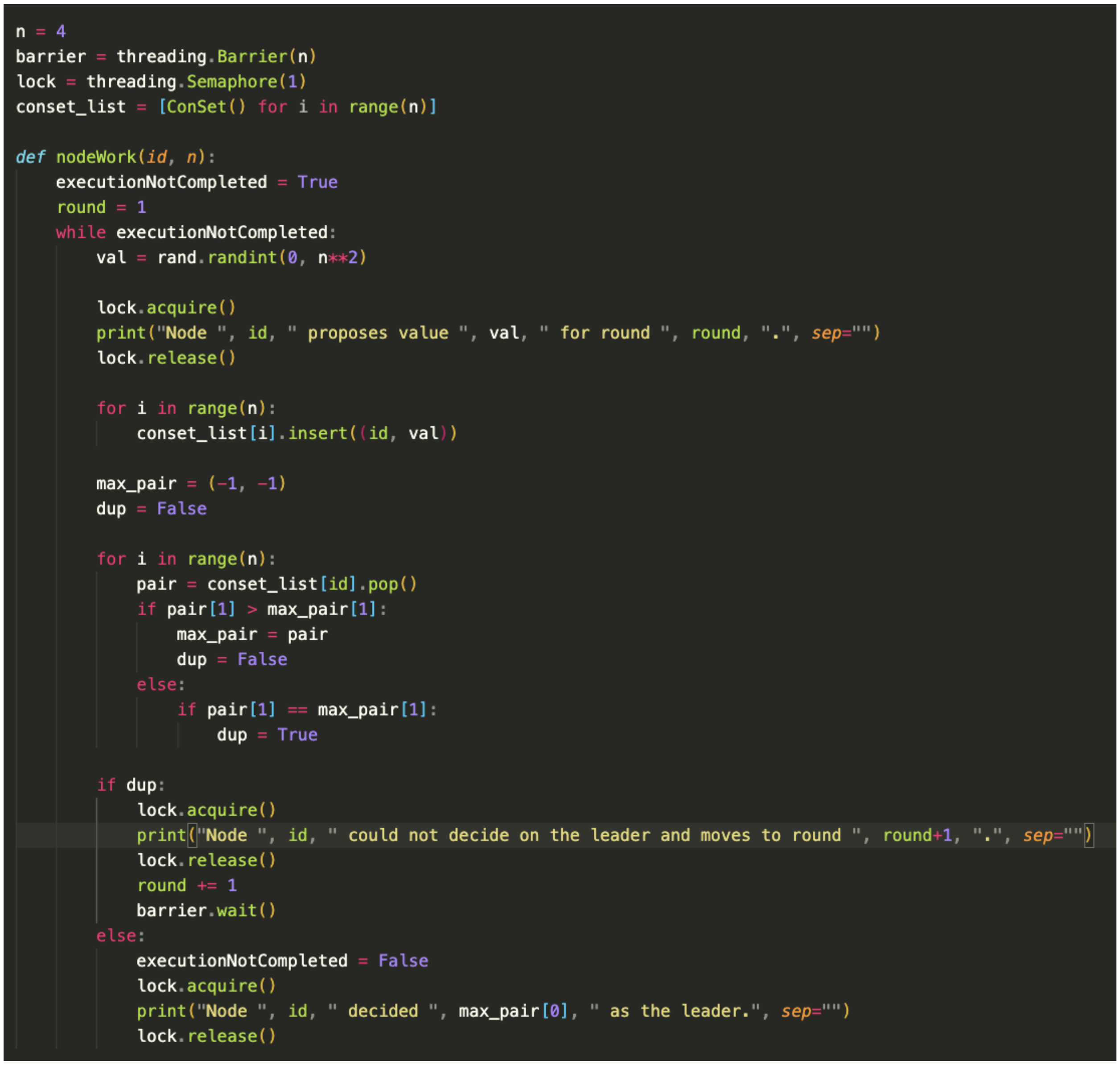


```
import threading

You, yesterday | 1 author (You)
class ConSet:
    def __init__(self):
        self.items = {}
        self.mutex = threading.Semaphore(1)
        self.cond = threading.Semaphore(0)

    def insert(self, newItem):
        self.mutex.acquire()
        if (newItem not in self.items) or self.items[newItem] == False:
            self.items[newItem] = True
            self.cond.release()
        self.mutex.release()
    def pop(self):
        self.cond.acquire()
        self.mutex.acquire()
        res = None
        for item in self.items:
            if self.items[item] == True:        You, yesterday • LAB1, HW1,
                self.items[item] = False
                res = item
                break
        self.mutex.release()
        return res
    def printSet(self):
        self.mutex.acquire()
        print("Cons Dict is", self.items)
        self.mutex.release()
```

I created one lock and one condition variable using Semaphores. Whenever a successful insertion operation happens I increment the condition variable by one. The number in the condition variable shows how many elements I have in the set. Also, I protect the insertion operation by using lock.

The pop operation first acquires the condition variable. If there are no elements in the set, it cannot acquire the condition variable and starts waiting. If there are elements in the set the condition variable should be bigger than 0, so it can acquire the condition variable and decrease the condition variable by 1. After acquiring the condition variable, it needs to lock the operation.

Leader.py



```python
n = 4
barrier = threading.Barrier(n)
lock = threading.Semaphore(1)
conset_list = [ConSet() for i in range(n)]

def nodeWork(id, n):
    executionNotCompleted = True
    round = 1
    while executionNotCompleted:
        val = rand.randint(0, n**2)

        lock.acquire()
        print("Node ", id, " proposes value ", val, " for round ", round, ".", sep="")
        lock.release()

        for i in range(n):
            conset_list[i].insert((id, val))

        max_pair = (-1, -1)
        dup = False

        for i in range(n):
            pair = conset_list[id].pop()
            if pair[1] > max_pair[1]:
                max_pair = pair
                dup = False
            else:
                if pair[1] == max_pair[1]:
                    dup = True

        if dup:
            lock.acquire()
            print("Node ", id, " could not decide on the leader and moves to round ", round+1, ".", sep="")
            lock.release()
            round += 1
            barrier.wait()
        else:
            executionNotCompleted = False
            lock.acquire()
            print("Node ", id, " decided ", max_pair[0], " as the leader.", sep="")
            lock.release()
```

The variable n gives the number of threads and the number of ConSets.
I used a barrier for this assignment since when the leader cannot be decided the threads should be blocked before starting the new round.

In the nodeWork function, I need two variables, id which is unique for each thread, and n which is the number of threads. My function has a while loop since in case of a tie in leader selection, the functionality should start over.

In the first part of my while loop. I found a value for the current node and printed it. After, I insert the id, and value pair for all of the ConSet objects in the global list. After the insertion, the popping starts. In each round, I compare the value of the popped element with the current maximum and update the current maximum if necessary. Also, I am detecting the ties in this part of the code.

In the end, I am either deciding the leader and finishing the execution or starting for another round.