Module 13: The Dining Philosopher's Problem

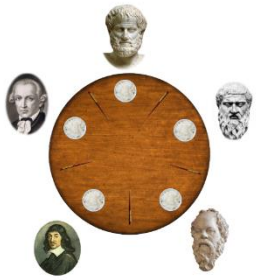
JHU EP 606.206 - Introduction to Programming Using Python

Introduction

In this assignment you will use Python's 'threading' library to use a synchronization primitive call a "lock" to solve the Dining Philosopher's Problem. The Dining Philosopher's Problem is a classic Computer Science exercise to demonstrate the need for locking mechanisms in multithreaded code. Without the use of a lock, or another synchronization primitive such as a semaphore, your solution to this problem may result in a "deadlock". For example, a chicken cannot exist without having first hatched from an egg. An egg cannot exist without first having been laid by a chicken. Therefore, in theory, a chicken cannot exist without en egg which cannot exist without a chicken. This scenario shows how each entity (chicken and egg) is dependent on something that it is not guaranteed to be able to obtain, thus, it is in a state called "deadlock." This week we'll see how to use locks and threads in Python to avoid a deadlock.

Python skills: classes, loops, list comprehensions, threads, synchronization primitives (locks)

The Dining Philosopher's Problem



There are 5 (silent) philosophers sitting around a circular dinner table, each with a plate of food in front of them and a fork in between each plate. To be able to eat a philosopher must pick up their left and right forks, if available. Each philosopher can be in one of 3 states:

1. Thinking: a philosopher is pondering and not currently interested in eating
2. Eating: a philosopher has acquired their left and right fork and is currently consuming food
3. Hungry: the philosopher is hungry but is waiting for one, or both, forks to become available

Each philosopher spends an arbitrary part of their time either thinking or eating and two philosophers cannot communicate or know what the other is thinking. Once a philosopher is done eating they will place both forks back on the table making them available for use by other philosophers. You can assume there is an infinite amount of food on each plate and that each philosopher will eventually become hungry again once they've eaten. **The problem is to design a concurrent algorithm so that no philosopher will starve (i.e. avoid a "deadlock" where one or more philosophers can never eat).** A deadlock would occur, for example, if all 5 philosophers sat down and all picked up their right fork. All philosophers would be waiting for their left fork to become available which will never happen. So, we need to ensure a philosopher will never pick up only one fork if it is available; they will only pick up their pair of forks when they are hungry and both forks are available.

Fork Class (fork.py)

The Fork class define a philosopher's "fork". In this assignment an instance of Fork contains a 'threading.Lock()' object. Recall from lecture that a Lock() object is a synchronization primitive that ensures a shared resource is only accessible by one object at a time. So, in order to make sure an instance of our "Fork" class can only be held by one philosopher at a time we've added a Lock() to it. To pick up a "Fork", a philosopher must try to 'acquire()' it. If the "Fork" is already held by another philosopher who is currently eating the call to 'acquire() will return False, otherwise it will return True and the "Fork" will be locked from other philosophers accessing it.

You will implement the following Fork API:

```
import threading

class Fork:
    def __init__(self):
      # add a lock as an instance variable

    def acquire_fork(self):
        # return True if you can acquire self.lock, False otherwise

    def release_fork(self):
        # release lock
```

Philosopher Class (philosopher.py)

The Philosopher class represents a single philosopher. It is a descendant of the Python 'threading.Thread' class so it inherits all of the methods from 'threading.Thread'. It also defines what it means for a philosopher to "think" and "eat". You will implement the following API:

```
import random
import threading
import time

from fork import Fork

class Philosopher(threading.Thread):
    running = True

    # initialize a Philosophers name, left fork, and right fork
    def __init__(self, name: str, left_fork: Fork, right_fork: Fork):
        # call 'threading' superclass constructor
        # initialize 'name' instance variable
        # initialize 'left_fork' instance variable
        # initialize 'right_fork' instance variable

    # run() is called by thread's start() method; starts the thread running
    def run(self):
        while self.running:
            # call think()
            # call eat()
            # print <Philosopher name> is cleaning up.

    # make philosopher think for a random number of seconds until hungry
    def think(self):
        # 'thinking' = random number of seconds between 3 and 5 using
        # random.uniform()
        # print <Philosopher name> is thinking for 'thinking' seconds.
        # sleep for 'thinking' seconds
        # print <Philosopher name> is now hungry.

    # make philosopher eat for a random number of seconds until thinking again
    def eat(self):
        # try to acquire left fork
            # if successful, try to acquire right fork
                # if successful
                    # 'eating' = random num of seconds between 3 and 5 using
                    # random.uniform()
                    # print <Philosopher name> is eating for 'eating' seconds.
                    # sleep for 'eating' seconds

                    # release right fork
                    # print <Philosopher name> has put down his right fork.

            # release left fork
            # print <Philosopher name> has put down his left fork.

            # else: return
```

Dining Philosophers Client (DiningPhilosophers.py)

This Python script will serve as your main() method and will create your Philosopher's and Fork's as well as start the processing threads and shut them down. Your code should implement the following client:

```
import time

from fork import Fork
from philosopher import Philosopher

def DiningPhilosophers():
    # create array of 5 names: Plato, Aristotle, Buddha, Marx, and Nietzsche
    # use a list comprehension to create 5 Fork's
    # use a list comprehension to create 5 Philosopher's and correctly
    # assign each pair of forks to each philosopher

    # start all 5 Philosopher threads (should be non-blocking)

    # sleep for 10 seconds

    # set 'running' to False

    # exit all threads

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

Deliverables

Please submit the following:

1. Your “Fork” source code in a file called fork.py
2. Your “Philosopher” source code in a file called philosopher.py
3. Your client code in a file called DiningPhilosophers.py
4. A PDF containing a screenshot containing your output (example below)

Sample Output

```
Plato is thinking for 4.310435763305234 seconds.
Aristotle is thinking for 4.073368320579804 seconds.
Buddha is thinking for 4.919052173362567 seconds.
Marx is thinking for 4.688408725956356 seconds.
Nietzsche is thinking for 3.552629446534185 seconds.
Nietzsche is now hungry.
Nietzsche is eating for 3.400846950162624 seconds.
Aristotle is now hungry.
Aristotle is eating for 3.971779706371928 seconds.
Plato is now hungry.
Plato is thinking for 4.52099735184018 seconds.
Marx is now hungry.
Marx has put down his left fork.
Marx is thinking for 3.145493729835688 seconds.
Buddha is now hungry.
Buddha is thinking for 4.0193673963050465 seconds.
Nietzsche has put down his right fork.
Nietzsche has put down his left fork.
Nietzsche is thinking for 3.281516548095256 seconds.
Marx is now hungry.
Marx is eating for 4.000457120493939 seconds.
Aristotle has put down his right fork.
Aristotle has put down his left fork.
Aristotle is thinking for 3.052346724663087 seconds.
Plato is now hungry.
Plato is eating for 3.497625998733225 seconds.
Buddha is now hungry.
Buddha has put down his left fork.
Buddha is thinking for 4.352674821090391 seconds.
Nietzsche is now hungry.
Nietzsche is cleaning up.
Aristotle is now hungry.
Aristotle is cleaning up.
Marx has put down his right fork.
Marx has put down his left fork.
Marx is cleaning up.
Plato has put down his right fork.
Plato has put down his left fork.
Plato is cleaning up.
Buddha is now hungry.
Buddha is eating for 3.7039813615905555 seconds.
Buddha has put down his right fork.
Buddha has put down his left fork.
Buddha is cleaning up.
```