Scott Schumacher

Program_2: dpa.py

8 November 2016

The Dining Philosophers Problem: Four threads (the philosophers) must share resources (the spaghetti). To do so, they must acquire locks (the forks). In so doing, they deny the other philosophers the opportunity to eat. One approach to the problem is ResourceWarning hierarchy solution, wherein each philosopher is required to acquire the forks in a fixed order, which helps prevent deadlock.

This program implements a sort of arbitration; the philosophers maintain a record of their bites and the Fork Class, which previously had no decison- making code, now limits the number of times ach philosopher may use a fork pair. Under true multi-threading, odd- and even- numbered philosophers would be able to eat in turn... philosophers 2 and 4 would be able to eat, then philosophers 1 and 3, usv. But since Python only allows a single thread at a time to run, we see here that each philosopher is limited to 50 bites, and then passes the forks to the next philosopher.

The order in which they eat is not fixed; it is random. But the imposition of even a little bit of order is enough to ensure equal distribution of the spaghetti.

Layout of the table (P = philosopher, f = fork): P0 f3 f0 P3 P1 f2 f1 P2 Number of philosophers at the table. There'll be the same number of forks.

"""==="""

```
import time
import threading
import random
import json
import struct


philosophers = []
forks = []
screenLock = threading.Lock()
numPhilosophers = 4



class Philosopher(threading.Thread):
```

```
    def __init__(self, index):
        threading.Thread.__init__(self)
        self.index = index
        self.count = 0


    """================================================================================
     A Philosopher could eat forever, but has to reset his count after 50 bites;
     during the time it takes to reset self.count to zero, another thread jumps
     in. The regulation and distribution of resources is a cooperative one; the
     threads report to the ForkPair class's Pickup function how many bites (locks)
     they have consumed, and the pickup function limits them to 50. In order to
     continue the thread has to reset its count to zero; a ten millisecond pause
     at that time ensures that another thread has time to acquire the lock.
     ================================================================================"""

    def run(self):
        # Assign left and right fork
        leftForkIndex = self.index
        rightForkIndex = (self.index - 1) % numPhilosophers
        forkPair = ForkPair(leftForkIndex, rightForkIndex)
        while True:
            forkPair.pickUp(self.count)
            self.count += 1
            # in lieu of graphics, print running score of who eats and how much:
            print('philosopher ', self.index, 'pickup; ', 'count: ', self.count)
            forkPair.putDown()
            if self.count > 50:
                self.count = 0
                time.sleep(10)


class ForkPair:

    def __init__(self, leftForkIndex, rightForkIndex):
        # Order forks by index to prevent deadlock
        if leftForkIndex > rightForkIndex:
            leftForkIndex, rightForkIndex = rightForkIndex, leftForkIndex
        self.firstFork = forks[leftForkIndex]
        self.secondFork = forks[rightForkIndex]


    """================================================================================
    PickUp limits the number of times that a thread can control the lock. As long
    as a thread's count is less than fifty, it may continue. Pickup has been
    expanded to accept count, the thread's self.count, as an argument. In a very
    primitive way, this arrangement resembles a monitor... one process is allowed
    in at time, and count is used as a condition variable.
    ================================================================================"""

    def pickUp(self, count):
        if count <= 50:
```

```
            self.firstFork.acquire()
            self.secondFork.acquire()

    def putDown(self):
        if self.firstFork.locked() and self.secondFork.locked():
            self.firstFork.release()
            self.secondFork.release()



if __name__ == "__main__":

    print('starting')

    # Create philosophers and forks
    for i in range(0, numPhilosophers):
        philosophers.append(Philosopher(i))
        forks.append(threading.Lock())


    # All philosophers start eating
    for philosopher in philosophers:
        philosopher.start()

    """
    # Allow CTRL + C to exit the program
    try:
        while True: time.sleep(.21)
    except (KeyboardInterrupt, SystemExit):
        os._exit(0)
    """
```