Parallel Programming Homework Report

Saygın Efe Yıldız
200316050

Introduction

This report is for the assignment given in the Parallel Programming course given by Bora Canbula [1]. The assignment is to find a solution to the dining philosopher's problem and implement it in Python using threads and locks.

Literary Analysis

In this report, the algorithm that will be used is the Lehmann and Rabin's solution [2]. The Lehmann and Rabin's solution was first conceived in the paper given in the References section in 1981. The paper first shows that a deterministic solution to the dining problem with distributed processes is not possible in section 4 of the paper. A solution is given that utilizes randomness to be able to give a solution to the dining philosopher's problem using distributed processes.

Free Philosopher's Algorithm

The Free Philosopher's Algorithm given in section 5 of the paper is an algorithm that utilizes randomness to achieve a distributed solution to the dining philosophers problem. The algorithm goes as thus:

1. The philosopher picks from the left or right chopstick randomly. When the chopstick is available, the philosophers lift the chopstick up.
2. The philosopher checks whether second chopstick is available.
   - 2.1. If the second chopstick is available the philosopher picks up the second chopstick and starts eating.
   - 2.2. If the second chopstick is not available the philosopher drops the first chopstick down and starts from step 1.
3. The philosopher eats the food.
4. The philosopher drops both chopsticks and starts again from step 1.

The randomness of this algorithm is what breaks the symmetry of the discrete algorithms that kept the distributed processes from working. This algorithm avoids deadlock but it does not avoid lockout. Lockout is still possible in the Free Philosopher's Algorithm because the same philosophers can keep picking up the same chopsticks leaving the other philosophers starving. Whilst the other philosophers will eventually get to eat, the processing (eating) done by the philosophers is not symmetrical.

The Courteous Philosopher's Algorithm

The Courteous Philosopher's Algorithm given in section 7 of the paper is the algorithm that addresses the lockout problem of the aforementioned algorithm. It does this by introducing mechanism that make the system more fair to the philosophers my making the philosophers more "courteous" so to speak.

The courteous philosopher's algorithm uses two types of variables to allow the philosophers to make more informed decisions. The variables are defined like this:

- The left-signal and right-signal variables are the variables that correspond to whether the philosopher is currently attempting to eat or not. The left-signal is shared with the left neighbor and the right-signal is shared with the right signal. The philosopher that the signals belong to can write to the signals to change them.
- The left-neighbor-signal and right-neighbor-signal are the same variables as the one above but from the neighboring philosopher's side. The left-neighbor-signal is the right-signal of the left neighbor and the right-neighbor-signal is the left-signal of the right neighbor. Since these signals are owned by the neighboring philosopher, the current philosopher cannot write to the variables to change them. For the current philosopher, the variables are read-only.
- The left-last and right-last variable is the shared variable that corresponds to which out of the neighbors ate last. This variable is shared by both of the philosophers and is updated based on which one ate last.

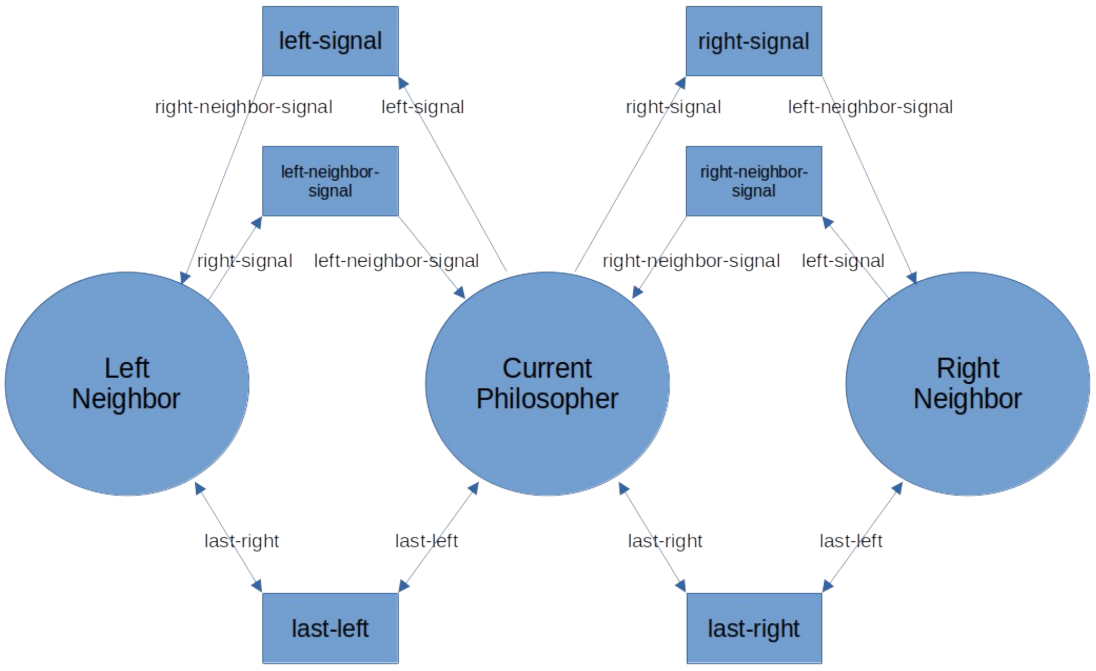


Figure 1. Graphic showing the variables in a more intuitive way.

The aforementioned variables will be used in the courteous philosopher's algorithm. The courteous philosopher's algorithm is defined as thus:

1. The philosopher turns on its left and right signals indicating that it is attempting to eat.
2. The philosopher picks from the left and right chopsticks randomly. The philosopher picks up the chopstick when it is available and if:
    2.1. The neighboring philosopher's signal is off indicating that he isn't also attempting to eat.
    2.2. The neighboring philosopher's was the last one out of both of them to eat or none of them ate yet.
3. The philosopher checks whether second chopstick is available.
    3.1. If the second chopstick is available the philosopher picks up the second chopstick and starts eating.
    3.2. If the second chopstick is not available the philosopher drops the first chopstick down and starts from step 1.
4. The philosopher eats the food.
5. The philosopher updates its left-last and right-last signals to indicate that it ate last.
6. The philosopher updates its left and right signal to signal that the philosopher is not attempting to eat anymore.
7. The philosopher drops both chopsticks and starts again from step 1.

Methodology

The algorithm was implements using Python 3.10.12 version in a virtual environment. The implementation is based on Bora Canbula's implementation which is the "diningphilosophers.py" located in the Week11 folder of the course's Github page [3]. There are two implementations, the "diningphilosophers_free.py" and the "diningphilosophers_courteous.py".

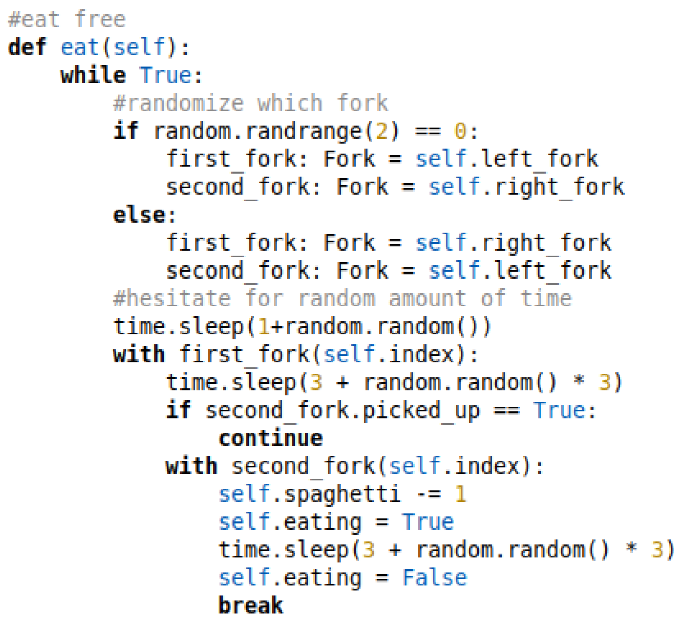
diningphilosophers_free.py

The "diningphilosophers_free.py" implementation is the implementation of the Free Philosopher's Algorithm. The implementation differs from Bora Canbula's implementation with the following changes:

- The "eat" method of the "Philosopher" class is changed to the new algorithm

eat method

- The whole method is contained in a while loop to be able to exit the with statement of the first lock when dropping the first fork prematurely.
  - A fork is randomly chosen from the left and right forks to be the first and second fork.
  - A sleep with a random time delay is introduced before picking up the first fork to not only increase randomness but to also to avoid busy waiting and overloading the CPU.
  - The first fork is picked up.
  - If the second fork has been picked up reset the while loop exiting the first fork's with statement releasing the lock.
  - The second fork is picked up.
  - Food is eaten



```python
#eat free
def eat(self):
    while True:
        #randomize which fork
        if random.randrange(2) == 0:
            first_fork: Fork = self.left_fork
            second_fork: Fork = self.right_fork
        else:
            first_fork: Fork = self.right_fork
            second_fork: Fork = self.left_fork
        #hesitate for random amount of time
        time.sleep(1+random.random())
        with first_fork(self.index):
            time.sleep(3 + random.random() * 3)
            if second_fork.picked_up == True:
                continue
            with second_fork(self.index):
                self.spaghetti -= 1
                self.eating = True
                time.sleep(3 + random.random() * 3)
                self.eating = False
                break
```

Figure 2. diningphilosophers_free.py eat method

diningphilosophers_courteous.py

The “” implementation is the implementation of the The Courteous Philosopher’s Algorithm. The implementation different from Bora Canbula’s implementation with the following changes:

- The “eat” method of the “Philosopher” class is changed to the new algorithm”
- The “last_picked_up_index” attribute is added to the “Fork” class.
- The “trying” attribute is added to the “Philosopher” class
- The “left_philosopher” and “right_philosopher” attributes are added to the class.

last_picked_up_index attribute

The “last_picked_up_index” attribute corresponds to the left-last and right-last variables in The Courteous Philosopher’s Algorithm. Since the “Fork” class was already shared between the philosophers I used stored the information about which philosopher ate last in the fork as to mean “who used this fork last”. Instead of storing the information as “left” or “right” the variable stores the index of the philosopher that last used it.

trying attribute

The “trying” attribute corresponds to the left-signal and right-signal variables in The Courteous Philosopher’s Algorithm. Since the left-signal and right-signal is owned by the Philosopher I thought I would add it as a class attribute of the “Philosopher” class. I added the trying as a separate attribute even though the eating attributes already exists. The reason why I still added the trying attribute is because the trying attributes encompasses the entire eating attempt whilst the eating attribute encompasses only when actually being able to eat. The trying attribute

left_philosopher and right_philosopher attributes

The “left_philosopher” and “right_philosopher” attributes represent the left neighboring philosopher and right neighboring philosopher of the current philosopher. The left-neighbor-signal and right-neighbor-signal variables are used in The Courteous Philosopher’s Algorithm and to be able to retrieve the signals from the neighboring philosophers the current philosopher needed to be able to access the state of the neighboring signals.

eat method

- The “trying” attribute is set to true to indicate to other philosophers that the current philosopher is eating.
- The whole method is contained in a while loop to be able to exit the with statement of the first lock when dropping the first fork prematurely.
  - A sleep with a random time delay is introduced before picking up the first fork to not only increase randomness but to also to avoid busy waiting and overloading the CPU.
  - A fork is randomly chosen from the left and right forks to be the first and second fork.
  - If the philosopher that the chosen fork is shared with is also trying to eat, then reset the while loop
  - If the fork was picked up by us previously, then reset the while loop.
  - The first fork is picked up.
  - If the second fork has been picked up reset the while loop exiting the first fork’s with statement releasing the lock.
  - The second fork is picked up.
  - Food is eaten.
- The “trying” attribute is set to false to indicate to other philosophers that the current philosopher is done eating.

```
#eat courteous
def eat(self):
    self.trying = True
    while True:
        #hesitate for random amount of time
        time.sleep(random.random())
        #randomize which fork
        if random.randrange(2) == 0:
            first_fork: Fork = self.left_fork
            second_fork: Fork = self.right_fork
            if self.left_philosopher.trying:        #type: ignore
                continue
        else:
            first_fork: Fork = self.right_fork
            second_fork: Fork = self.left_fork
            if self.right_philosopher.trying:        #type: ignore
                continue
        #check
        if first_fork.last_picked_up_index == self.index:
            continue
        with first_fork(self.index):
            time.sleep(3 + random.random() * 3)
            if second_fork.picked_up == True:
                continue
            with second_fork(self.index):
                first_fork.last_picked_up_index = self.index
                second_fork.last_picked_up_index = self.index
                self.spaghetti -= 1
                self.eating = True
                time.sleep(3 + random.random() * 3)
                self.eating = False
                break
    self.trying = False
```

Figure 3. diningphilosophers_courteous.py eat method

Conclusion

In this paper, he looked at a solution of the dining philosopher's problem using the Lehmann and Rabin's solution. We looked at how the solution specified in the paper could be implemented in Python. The algorithm shown here and allows for a nice mechanism for managing shared resources. The algorithm presented here is one of many algorithms that can be used for this purpose and there are definitely other alternatives that work just as well or even better.

References

[1] https://github.com/canbula/ParallelProgramming

[2] Daniel Lehmann and Michael O. Rabin. 1981. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '81). Association for Computing Machinery, New York, NY, USA, 133–138. https://doi.org/10.1145/567532.567547

[3] https://github.com/canbula/ParallelProgramming/tree/0aa29fa6695b02846686b9bf692b0df20aed8628/Week11