Experiment 07- To write a program to implement the dining philosopher's problem in python

Learning Objective: Students should be able to understand the dining philosopher's problem by using different coding languages, such as C/C++/Java/Python.

Tools: Online compiler

Theory: The Dining Philosophers Problem is a classic synchronization problem in computer science that illustrates issues related to resource allocation and process synchronization in an operating system.

Problem Statement: Five philosophers sit around a circular table with a fork placed between each pair. Each philosopher must alternate between thinking and eating. However, to eat, a philosopher needs to pick up both the left and right forks. Since there are only five forks, contention arises when multiple philosophers try to eat at the same time.

Challenges: Deadlock: If every philosopher picks up their left fork simultaneously, they will all wait indefinitely for the right fork.

Starvation: Some philosophers may never get a chance to eat if others continuously acquire and release forks.

Solution Approach: This experiment implements the dining philosopher's problem using **multithreading in Python** with the `threading` module and a **Monitor-based approach** to avoid deadlock and starvation. Each philosopher is represented as a thread and must acquire locks (forks) to proceed with eating.

The monitor:

- Maintains the state of each philosopher (`THINKING`, `HUNGRY`, `EATING`).
- Ensures that a philosopher can eat only if both neighbors are not eating.
- Uses condition variables for synchronization.

This ensures that each philosopher gets a fair chance to eat, avoiding both **deadlock** and **starvation**.

Code: import threading

import time

N = 10,THINKING = 2,HUNGRY = 1,EATING = 0,times = 200

phil = [i for i in range(N)]

```
class Monitor:
    def __init__(self):
        self.state = [THINKING] * N
        self.phcond = [threading.Condition() for _ in range(N)]
    def test(self, phnum):
        with self.phcond[phnum]:
            if (self.state[(phnum + 1) % N] != EATING and
                    self.state[(phnum + N - 1) % N] != EATING and
                    self.state[phnum] == HUNGRY):
                self.state[phnum] = EATING
                self.phcond[phnum].notify()
    def take_fork(self, phnum):
        with self.phcond[phnum]:
            # Indicates it is hungry
            self.state[phnum] = HUNGRY
            self.test(phnum)
            # If unable to eat, wait for the signal
            if self.state[phnum] != EATING:
                self.phcond[phnum].wait()
            print(f"Philosopher {phnum} is Eating")
    def put_fork(self, phnum):
        with self.phcond[phnum]:
            # Indicates that the philosopher is thinking
            self.state[phnum] = THINKING
```

```
33      # Take Fork function
34      def take_fork(self, phnum):
35          with self.phcond[phnum]:
36              # Indicates it is hungry
37              self.state[phnum] = HUNGRY
38              # Test for condition
```

Ln: 50, Col: 1

Run | Share | $ | Command Line Arguments

```
** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Learning Outcomes: The student should have the ability to:

LO2.1 Outline various compilers for different languages
LO2.2 Understood the dining philosopher problem
LO2.3 Choose an appropriate compiler to solve the dining philosopher solution

Course Outcomes: Upon completion of the course, students will be able to learn about operating systems and security concepts.

Conclusion: The implementation of the **Dining Philosophers Problem** in Python using **multithreading and monitors** helps us understand process synchronization and resource allocation in operating systems. By using condition variables and ensuring that a philosopher eats only when both neighbors are not eating, we successfully **prevent deadlock and starvation**. This experiment demonstrates the importance of synchronization techniques in concurrent programming.

For Faculty Use:

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance/ Learning Attitude [20%]**	
Marks Obtained				