**Batch: A-3 Roll No.: 16010122104**

**Experiment No. 06**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **TITLE:** Implementation of dining philosopher problem using threads. |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**AIM:** Implementation of Process synchronization algorithms using threads – Dining Philosopher problem

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

**CO 2.** To understand the concept of process, thread and resource management.

**CO 3.** To understand the concepts of process synchronization and deadlock.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. **Silberschatz A., Galvin P., Gagne G. “Operating Systems Principles”, Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate “Operating Systems” McGraw Hill Third**

**Edition.**

1. **William Stallings, “Operating System Internal & Design Principles”, Pearson.**
2. **Andrew S. Tanenbaum, “Modern Operating System”, Prentice Hall.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pre Lab/ Prior Concepts:**

Knowledge of Concurrency, Mutual Exclusion, Synchronization, Deadlock, Starvation,threads.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Description of the chosen process synchronization algorithm:**

The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

**Implementation details:** (printout of code)

import threading

import time

import random

num\_philosophers = 5

max\_meals = 3

forks = [threading.Semaphore(1) for \_ in range(num\_philosophers)]

mutex = threading.Lock()

meals\_eaten = [0] \* num\_philosophers

def philosopher(id):

    global meals\_eaten

    while True:

        if meals\_eaten[id] >= max\_meals:

            print(f"Philosopher {id} has finished eating {max\_meals} times and leaves the table.")

            break

        print(f"Philosopher {id} is thinking.")

        time.sleep(random.uniform(1, 3))

        with mutex:

            print(f"Philosopher {id} is hungry and trying to pick up forks.")

            forks[id].acquire()

            forks[(id + 1) % num\_philosophers].acquire()

            print(f"Philosopher {id} is eating.")

            time.sleep(random.uniform(1, 2))

            meals\_eaten[id] += 1

            forks[id].release()

            forks[(id + 1) % num\_philosophers].release()

            print(f"Philosopher {id} has finished eating and put down forks.")

threads = []

for i in range(num\_philosophers):

    t = threading.Thread(target=philosopher, args=(i,))

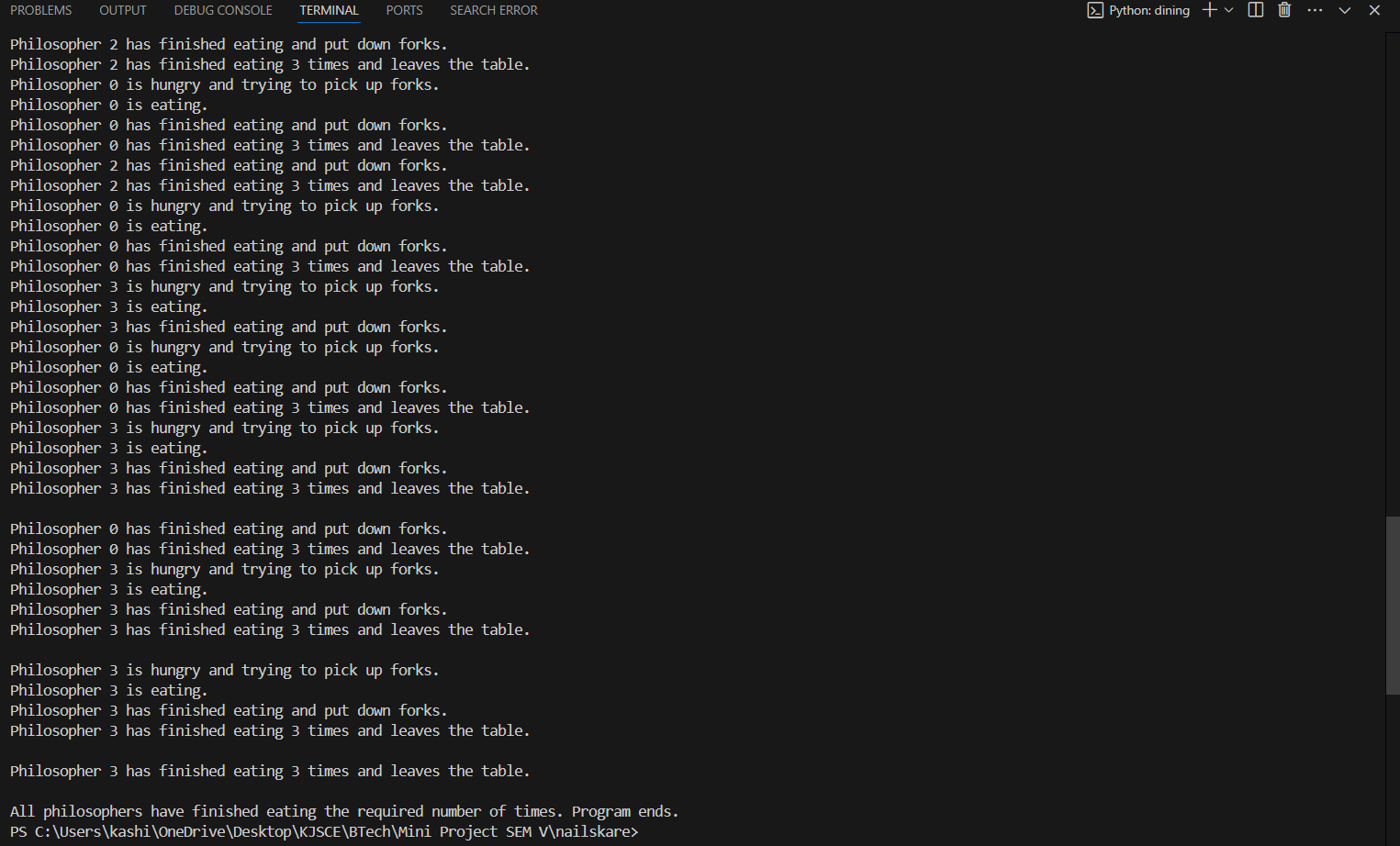
    threads.append(t)

    t.start()

for t in threads:

    t.join()

print("\nAll philosophers have finished eating the required number of times. Program ends.")



**Conclusion:**

We implemented of dining philosopher problem

**Post Lab Descriptive Questions**

1.Differentiate between a monitor, semaphore and a binary semaphore?

Ans:

Monitor:

A monitor is a high-level synchronization construct that allows threads to have both mutual exclusion and the ability to wait for a condition to become true.

Monitors encapsulate shared variables, procedures, and data structures, providing a mechanism for only one process to access a resource at a time.

It offers both locking and condition variable support, which allows threads to wait and signal each other.

Semaphore:

A semaphore is a lower-level synchronization mechanism represented by a counter.

It can have any integer value and is used for signaling between threads or processes.

Two primary operations, wait() (or P) and signal() (or V), control access.

It does not have the concept of ownership, so any thread can release the semaphore regardless of which thread acquired it.

Binary Semaphore:

A binary semaphore is a type of semaphore that can only have values 0 or 1.

It is commonly used as a mutex lock, ensuring mutual exclusion.

When set to 1, it indicates that the resource is free; when set to 0, it indicates that the resource is locked.

Similar to a general semaphore but limited to binary states, making it more straightforward to implement for simple locking.

2. Define clearly the dining-philosophers problem?

Ans:

The Dining Philosophers Problem is a classic synchronization problem that illustrates issues of resource sharing, deadlock, and starvation. It involves five philosophers seated around a table with five forks (or chopsticks) placed between them. Each philosopher alternates between two states: thinking and eating. To eat, a philosopher needs to pick up both the left and the right fork. However, only one philosopher can hold a fork at any time.

The challenge lies in devising a strategy that allows all philosophers to eat without causing deadlocks (where no philosopher can proceed) or starvation (where some philosophers never get a chance to eat).

3. Identify the scenarios in the dining-philosophers problem that leads to the deadlock situations?

Ans:

Deadlock occurs in the dining philosophers problem if each philosopher picks up one fork and waits indefinitely for the other fork. This situation can arise if:

* Each philosopher picks up their left fork simultaneously and waits for the right fork, resulting in a cycle of waiting.
* No philosopher can proceed to eat, as each is waiting for a fork held by their neighbor, creating a circular wait condition.

In this case, all philosophers are in a state of *deadlock*, unable to eat or release the forks they currently hold, leading to an indefinite wait for resources.

**Date: 25/10/2024 Signature of faculty in-charge**