A Course Based Project Report

On

DINING PHILOSOPHERS PROBLEM USING SEMAPHORES

Submitted in partial fulfillment of requirement for the completion of the

Operating Systems Laboratory

II B.Tech Computer Science and Engineering

of

VNR VJIET

By

B. NANDA KISHORE	**– 23071A05E6**
D. RISHI VARMA	**– 23071A05F1**
D. NAVADEEP	**– 23071A05F5**
G. KARTHIK	**– 23071A05F7**
U. HARADEEP	**– 23071A05K4**

2024-2025

VALLURIPALLI NAGESWARA RAO
VIGNANA JYOTHI INSTITUTE OF ENGINEERING &TECHNOLOGY
(AUTONOMOUS INSTITUTE)
NAAC ACCREDITED WITH 'A++' GRADE
NBA Accreditation for B. Tech Programs
Vignana Jyothi Nagar, Bachupally, Nizampet (S.O), Hyderabad 500090
Phone no: 040-23042758/59/60, Fax: 040-23042761
Email:postbox@vnrvjiet.ac.in Website: www.vnrvjiet.ac.in

A Project Report On

DINING PHILOSOPHERS PROBLEM USING SEMAPHORES

Submitted in partial fulfillment of requirement

for the completion of the

Operating Systems Laboratory

II B.Tech Computer Science and Engineering

of

VNRVJIET

2024-2025

Under the Guidance

of

DR. D N VASUNDHARA

Assistant Professor

Department of CSE

VNR VIGNANA JYOTHI INSTITUTE OF ENGINEERING & TECHNOLOGY

(AUTONOMUS INSTITUTE)

NAAC ACCREDITED WITH ‘A++’ GRADE

CERTIFICATE

This is to certify that the project entitled “DINING PHILOSOPHERS PROBLEM USING SEMAPHORES” submitted in partial fulfillment for the course of Operating Systems laboratory being offered for the award of Batch (CSE-C) by VNRVJIET is a result of the bonafide work carried out by **23071A05E6**, **23071A05F1, 23071A05F5, 23071A05F7** and **23071A05K4** during the year **2024-2025**.

This has not been submitted for any other certificate or course.

.

Signature of Faculty **Signature of Head of the Department**

ACKNOWLEDGEMENT

An endeavor over a long period can be successful only with the advice and support of many well-wishers. We take this opportunity to express our gratitude and appreciation to all of them.

We wish to express our profound gratitude to our honorable **Principal Dr. C.D.Naidu** and **HOD Dr.V.Baby**, **CSE Department**, **VNR Vignana Jyothi Institute of Engineering and Technology** for their constant and dedicated support towards our career moulding and development.

With great pleasure we express our gratitude to the internal guide **DR. D.N VASUNDHARA, Assistant Professor, CSE department** for his timely help, constant guidance, cooperation, support and encouragement throughout this project as it has urged us to explore many new things.

Finally, we wish to express my deep sense of gratitude and sincere thanks to our parents, friends and all our well-wishers who have technically and non-technically contributed to the successful completion of this course-based project.

DECLARATION

We hereby declare that this Project Report titled "**MULTI THREADED PROXY SERVER**" submitted by us of Computer Science & Engineering in **VNR Vignana Jyothi Institute of Engineering and Technology**, is a bonafide work undertaken by us and it is not submitted for any other certificate /Course or published any time before.

Name & Signature of the Students:

B. NANDA KISHORE – 23071A05E6

D. RISHI VARMA – 23071A05F1

D. NAVADEEP – 23071A05F5

G. KARTHIK – 23071A05F7

U. HARADEEP – 23071A05K4

Date:

TABLE OF CONTENTS

ABSTRACT

The Dining Philosophers Problem is a classic synchronization challenge that demonstrates the difficulties of resource sharing and process coordination in concurrent programming.

This project aims to visualize the problem using two synchronization techniques—semaphores and monitors—to help users understand how to avoid issues like deadlock, race conditions, and starvation. Each philosopher is simulated as a thread that alternates between thinking and eating, requiring access to two shared forks (resources) placed between them. The project offers a visual and interactive platform to demonstrate how the philosophers coordinate access to these forks while maintaining safe and efficient behavior.

The visualization illustrates the behavior of each philosopher using intuitive color codes and animations, making the concept accessible and educational. In the semaphore-based model, the program uses binary semaphores for each fork and strategies such as limiting the number of concurrent eaters to avoid deadlock. In the monitor-based model, mutual exclusion and condition variables are used to ensure safe fork allocation.

The application allows users to toggle between both models, compare their behavior, and observe the effects of different concurrency control methods. This project serves as a practical tool for students and developers to better grasp operating system concepts and thread synchronization mechanisms.

INTRODUCTION

The Dining Philosophers Problem is a classic synchronization problem in computer science and operating systems that illustrates issues related to concurrency and resource sharing. It demonstrates the challenges of ensuring that multiple processes (in this case, philosophers) can share resources (forks) without causing deadlock or race conditions. The problem was introduced by E.W. Dijkstra in 1965. Imagine five philosophers sitting at a round table. Each philosopher has a plate of food and two forks (one on either side of them). In order to eat, each philosopher must use both forks. The philosophers alternate between thinking and eating. Here are the main constraints:

Each philosopher must pick up both forks to eat. They must put the forks down after eating. Philosophers cannot eat at the same time; they must use both forks simultaneously. The goal is to manage the philosophers' actions to avoid deadlock (where no philosopher can proceed because they are each holding one fork and waiting for the other) and starvation (where a philosopher is unable to eat indefinitely).

Key Problems:

1. Deadlock: This occurs if all philosophers pick up their left fork simultaneously and wait for the right one, resulting in all philosophers waiting indefinitely.
2. Starvation: This happens when one or more philosophers never get a chance to eat because others monopolize the forks.
3. Concurrency: Philosophers must operate concurrently, and the system should ensure that forks are shared without causing conflicts.

A semaphore is a low-level synchronization tool that controls access to shared resources. Semaphores are integer variables used to signal or block access to critical sections. In the Dining Philosophers problem, semaphores are used to represent forks and synchronize philosophers' access to them. Key Concepts: A binary semaphore (value 0 or 1) is used for each fork, where a value of 1 indicates that the fork is available, and 0 indicates that it is being used. A mutex semaphore is used to protect the critical section where philosophers try to pick up the forks.

METHODOLOGY

The development of the Dining Philosophers simulation using semaphores was carried out in a structured, step-by-step approach to ensure synchronization, deadlock prevention, and correctness. The methodology includes the design, implementation, and testing of the synchronization logic using semaphores to simulate concurrent philosopher threads accessing shared resources (forks). The process can be broken down into the following key steps:

•**Requirement Analysis and Design Planning**: The project began by identifying the fundamental requirements of the classic Dining Philosophers problem. These included representing each philosopher as a separate thread, modeling forks as shared resources, and preventing problems like deadlock and starvation. A modular approach was taken to separate core components like thread management, semaphore handling, and resource control. The design focused on mutual exclusion and fairness in resource allocation.

•**Thread and Semaphore Setup**: The next step was to implement POSIX threads (pthreads) to simulate philosophers as concurrent entities. Each philosopher thread represents an infinite loop of thinking and eating. Semaphores were initialized to control access to forks, ensuring only one philosopher could use a fork at a time. This setup provided the foundation for concurrent execution and synchronization.

•**Concurrency Control with Semaphores**: Semaphores played a key role in managing concurrency. Each fork was protected using a binary semaphore. Philosophers attempted to acquire both adjacent forks (left and right) before eating. If either fork was unavailable, the philosopher would wait. To avoid deadlock, techniques such as limiting the number of simultaneous eaters or imposing an order on fork acquisition were explored and implemented. This phase required careful management of critical sections and synchronization logic.

•**Philosopher Behavior Simulation**: A dedicated routine was developed to simulate each philosopher's life cycle: thinking, hungry, attempting to acquire forks, eating, and releasing forks. This routine ensured that state transitions were safe and atomic. Logging mechanisms were included to track actions and state changes in real-time, aiding in verification and debugging.

- **Testing and Optimization**: The system was tested with different numbers of philosophers and varied timing for eating and thinking to evaluate performance and correctness. Edge cases such as simultaneous fork requests were analyzed. Logs were reviewed to confirm deadlock-free operation and fair resource access. Code was refined for readability, efficiency, and modularity to support future enhancements or alternative synchronization techniques.

Through this step-by-step methodology, the project successfully demonstrates the practical application of operating system concepts like process synchronization, mutual exclusion, and thread communication. Each phase—from thread creation to semaphore handling—was carefully designed to simulate real-world concurrency issues. By managing shared resources with semaphores and ensuring proper synchronization, the system effectively models the challenges of concurrent computing. This structured approach not only validates theoretical principles but also enhances understanding of OS-level synchronization techniques.

OBJECTIVES

The main aim of this project is to build an interactive and educational tool that enables users to visualize and understand Dining Philosophers process management concepts effectively. Below are the core objectives:

Understand Concurrency Issues:

Learn how multiple processes (philosophers) can run concurrently, each trying to access shared resources (forks), and the challenges that arise from this, such as deadlock and starvation.

Explore Synchronization Mechanisms:

Gain an understanding of different synchronization techniques like semaphores and monitors. Learn how these synchronization tools can be used to control access to shared resources and prevent conflicts in concurrent systems.

Address Deadlock:

Investigate how to avoid deadlock, a situation where processes cannot proceed because they are each waiting for resources held by others. In the case of the dining philosophers, deadlock occurs when every philosopher holds one fork and waits for the other

Prevent Starvation:

Study how to design systems that avoid starvation, where some processes (philosophers) never get a chance to execute their critical section (eating), despite others repeatedly gaining access to the shared resources.

Design Scalable Systems:

Learn how to extend the principles of the Dining Philosophers Problem to design systems with many processes or resources, ensuring that such systems are scalable and free of synchronization issues like deadlock or starvation

FLOW OF EXECUTION

Start Program
|
V
Initialize System
- Create philosopher threads
- Initialize semaphores for each fork
- Initialize mutex (if required for synchronization control)
|
V
Philosopher Thread Begins
- Each philosopher starts an infinite loop:
-> Think
-> Become hungry
-> Try to pick up forks
|
V
Acquire Forks (Semaphore Wait)
- Wait (P) on left fork semaphore
- Wait (P) on right fork semaphore
- Ensure mutual exclusion using mutex (if needed)
|
V
Eat
- Philosopher eats for a fixed/random duration
- Critical section access is in progress
|
V
Release Forks (Semaphore Signal)
- Signal (V) left fork semaphore
- Signal (V) right fork semaphore
- Exit critical section
|
V
Repeat Cycle
- Go back to thinking
|
V
End Program (if simulation is terminated manually)

IMPLEMENTATION OF PROGRAM

Code:

```
import tkinter as tk
import threading
import time
import random
from math import cos, sin, radians

class Philosopher(threading.Thread):
    def _init_(self, canvas, forks, condition, index, position, size, spoon_positions, stop_event, log_text):
        threading.Thread._init_(self)
        self.canvas = canvas
        self.forks = forks
        self.condition = condition
        self.index = index
        self.position = position
        self.size = size
        self.state = "thinking"
        self.spoon_positions = spoon_positions
        self.stop_event = stop_event
        self.log_text = log_text
        self.head = canvas.create_oval(position[0] - size, position[1] - size,
                                       position[0] + size, position[1] + size,
                                       fill="white", outline="black", width=2)
        self.body = canvas.create_line(position[0], position[1] + size, position[0], position[1] + size + 30, width=2)
        self.left_arm = canvas.create_line(position[0], position[1] + size + 10, position[0] - 20, position[1] + size + 20, width=2, fill="black")
        self.right_arm = canvas.create_line(position[0], position[1] + size + 10, position[0] + 20, position[1] + size + 20, width=2, fill="black")
        self.left_leg = canvas.create_line(position[0], position[1] + size + 30, position[0] - 20, position[1] + size + 50, width=2, fill="black")
        self.right_leg = canvas.create_line(position[0], position[1] + size + 30, position[0] + 20, position[1] + size + 50, width=2, fill="black")
        self.text = canvas.create_text(position[0], position[1] + size, text=f'P{index+1}', fill="black", font=("Helvetica", 14, "bold"))
        self.forks_lines = []  # To keep track of fork lines
```

```
    def run(self):
        while not self.stop_event.is_set():
            self.think()
            self.hungry()
            self.eat()

    def think(self):
        if self.stop_event.is_set():
            return
        self.state = "thinking"
        self.update_canvas()
        self.log(f"Philosopher {self.index+1} is thinking")
        time.sleep(random.uniform(1, 3))

    def hungry(self):
        if self.stop_event.is_set():
            return
        self.state = "hungry"
        self.update_canvas()
        self.log(f"Philosopher {self.index+1} is hungry")
        # Asymmetric approach: pick the fork order depending on philosopher's index
        left_fork_index = self.index if self.index % 2 == 0 else (self.index + 1) % len(self.forks)
        right_fork_index = (self.index + 1) % len(self.forks) if self.index % 2 == 0 else self.index
        with self.condition:
            while self.forks[left_fork_index] or self.forks[right_fork_index]:
                self.condition.wait()
            self.forks[left_fork_index] = self.forks[right_fork_index] = True
        self.pick_forks(left_fork_index, right_fork_index)
```

```
def eat(self):
    if self.stop_event.is_set():
        return
    self.state = "eating"
    self.update_canvas()
    self.log(f"Philosopher {self.index+1} is eating")
    time.sleep(random.uniform(1, 3))
    left_fork_index = self.index if self.index % 2 == 0 else (self.index + 1) % len(self.forks)
    right_fork_index = (self.index + 1) % len(self.forks) if self.index % 2 == 0 else self.index
    with self.condition:
        self.forks[left_fork_index] = self.forks[right_fork_index] = False
        self.condition.notify_all()
    self.put_down_forks(left_fork_index, right_fork_index)
```

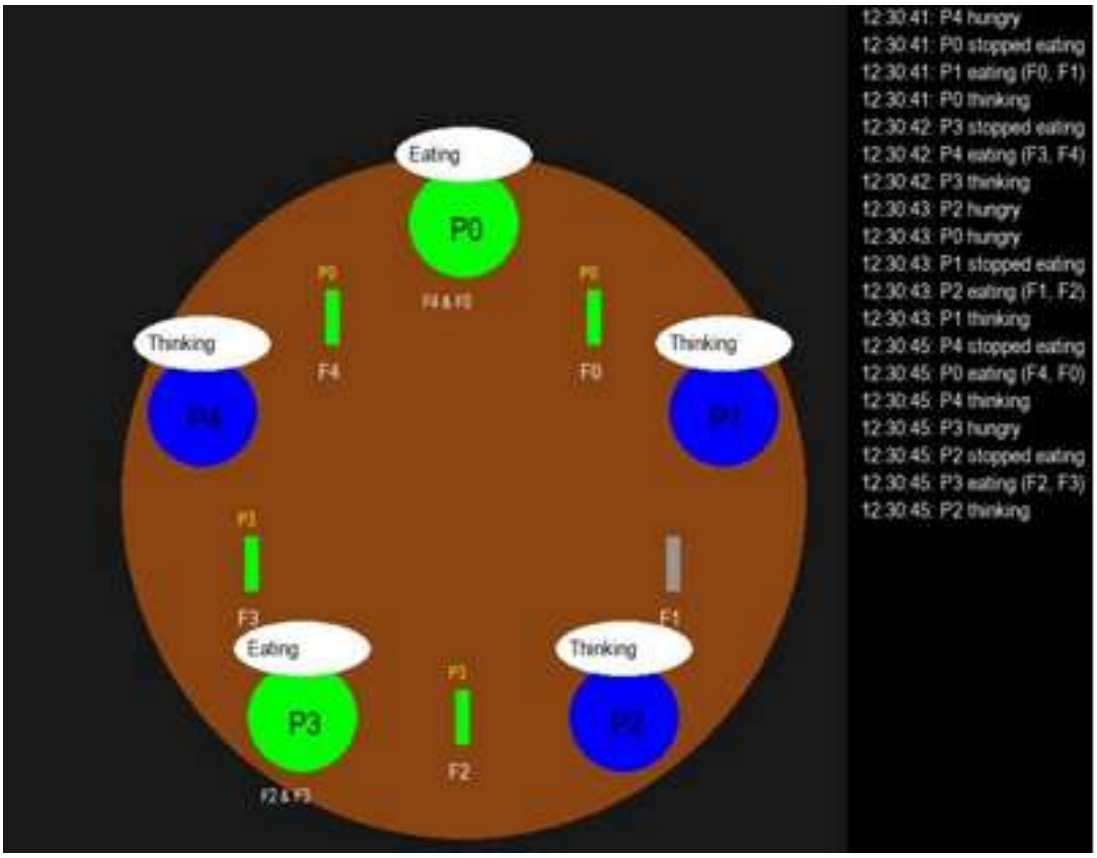
```python
def pick_forks(self, left_fork_index, right_fork_index):
    if self.stop_event.is_set():
        return
    # Draw fork lines when philosopher picks up forks, show which fork was picked first
    color = "blue" if self.index % 2 == 0 else "green"  # Blue for even index, Green for odd
    fork_line_left = self.canvas.create_line(self.position, self.spoon_positions[left_fork_index], fill=color, width=2)
    fork_line_right = self.canvas.create_line(self.position, self.spoon_positions[right_fork_index], fill=color, width=2)
    self.forks_lines.append(fork_line_left)
    self.forks_lines.append(fork_line_right)

def put_down_forks(self, left_fork_index, right_fork_index):
    if self.forks_lines:
        self.canvas.delete(self.forks_lines.pop())
        self.canvas.delete(self.forks_lines.pop())

def update_canvas(self):
    color = {"thinking": "lightgrey", "hungry": "orange", "eating": "green"}[self.state]
    # Philosopher's head (circle) changes color according to state
    self.canvas.itemconfig(self.head, fill=color)
    self.canvas.update()
```

```python
    def log(self, message):
        self.log_text.config(state=tk.NORMAL)
        self.log_text.insert(tk.END, message + "\n")
        self.log_text.yview(tk.END)
        self.log_text.config(state=tk.DISABLED)

class DiningPhilosophers:
    def _init_(self, root, num_philosophers):
        self.frame = tk.Frame(root)
        self.frame.pack(side=tk.LEFT)

        self.canvas = tk.Canvas(self.frame, width=800, height=600, bg="lightblue")
        self.canvas.pack(side=tk.LEFT)

        self.log_text = tk.Text(self.frame, width=50, height=38, state=tk.DISABLED, bg="lightgray", fg="black", font=("Helvetica", 10))
        self.log_text.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

        self.forks = [False] * num_philosophers  # False indicates fork is available
        self.condition = threading.Condition()
        self.philosophers = []
        self.stop_event = threading.Event()

        angle_step = 360 / num_philosophers
        radius = 250
        center = (400, 300)
        table_radius = 150
        self.canvas.create_oval(center[0] - table_radius, center[1] - table_radius,
                                center[0] + table_radius, center[1] + table_radius,
                                fill="burlywood")
```

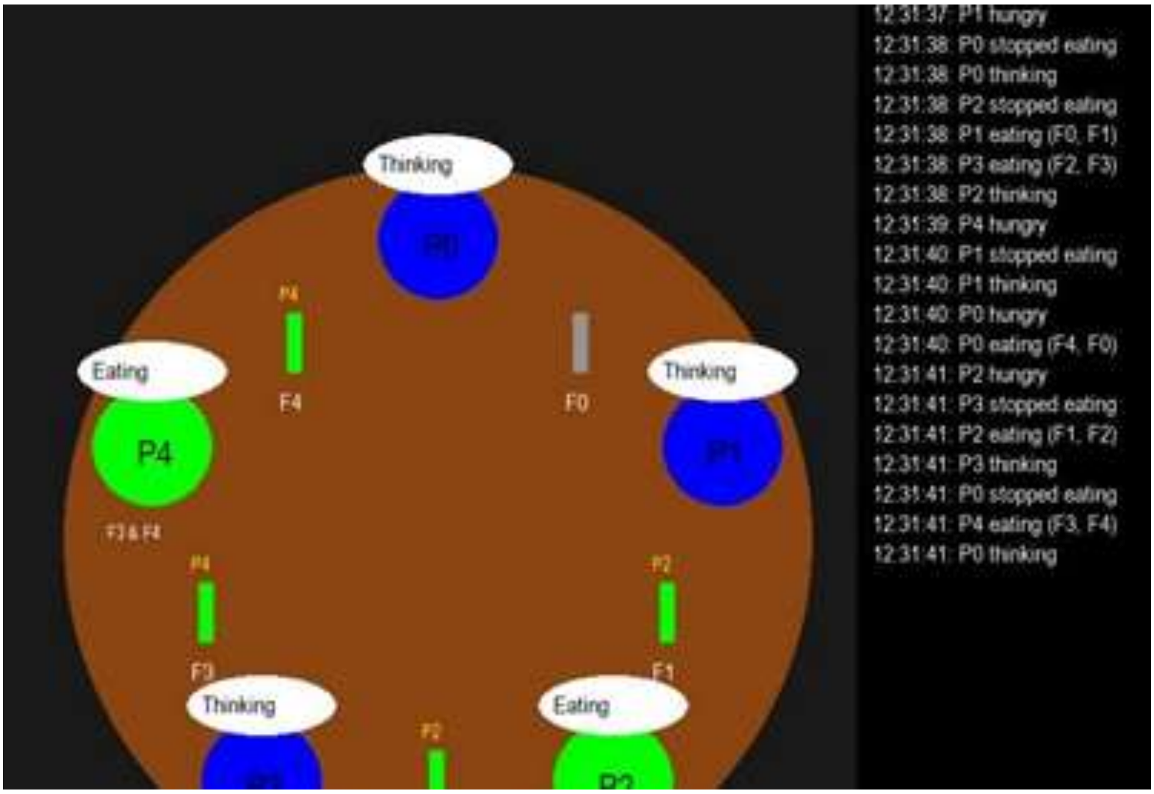
```python
spoon_radius = 120
self.spoon_positions = []
for i in range(num_philosophers):
    angle = angle_step * i
    x = center[0] + spoon_radius * cos(radians(angle))
    y = center[1] + spoon_radius * sin(radians(angle))
    self.spoon_positions.append((x, y))
    self.canvas.create_line(center[0], center[1], x, y, width=5, fill="grey")
    # Draw fork numbers for each fork
    self.canvas.create_text(x, y, text=f"{i+1}", fill="black", font=("Helvetica", 10))

for i in range(num_philosophers):
    angle = angle_step * i
    x = center[0] + radius * cos(radians(angle))
    y = center[1] + radius * sin(radians(angle))
    philosopher = Philosopher(self.canvas, self.forks, self.condition, i, (x, y), 30, self.spoon_positions, self.stop_event, self.log_text)
    self.philosophers.append(philosopher)

self.start_button = tk.Button(root, text="Start", command=self.start, font=("Arial", 12), bg="green", fg="white")
self.start_button.pack(side=tk.LEFT)

self.stop_button = tk.Button(root, text="Stop", command=self.stop, font=("Arial", 12), bg="red", fg="white")
self.stop_button.pack(side=tk.LEFT)
```

```
f start(self):
  self.stop_event.clear()
  for philosopher in self.philosophers:
      if not philosopher.is_alive():
          philosopher.start()
)
    def stop(self):
        self.stop_event.set()
        for philosopher in self.philosophers:
            philosopher.join()  # Ensure all threads have finished

def main():
    root = tk.Tk()
    root.title("Dining Philosophers Simulation")
    dp = DiningPhilosophers(root, 5)
    root.mainloop()

if _name_ == "_main_":
    main()
```

Output:



The above snapshot captures the dynamic activity of the semaphore solution, where philosophers have begun to contend for forks. Specifically, Philosopher 2 and Philosopher 5 have successfully transitioned into the eating state, denoted by green heads. Blue and green lines are used to visually represent the forks they've acquired, showing which philosopher picked which fork first (e.g., blue for even-indexed philosophers, green for odd-indexed). This visual also highlights how semaphore constraints allow non-adjacent philosophers to eat simultaneously, demonstrating concurrency without deadlock. Other philosophers, such as P3 and P4, remain in the thinking or waiting state. This diagram reflects how semaphore logic requires philosophers to acquire two forks in a specific order, enabling safe and fair resource sharing.



This snapshot captures the dynamic activity of the semaphore solution, where philosophers have begun to contend for forks. Specifically, Philosopher 2 and Philosopher 5 have successfully transitioned into the eating state, denoted by green heads. Blue and green lines are used to visually represent the forks they've acquired, showing which philosopher picked which fork first (e.g., blue for even-indexed philosophers, green for odd-indexed). This visual also highlights how semaphore constraints allow non-adjacent philosophers to eat simultaneously, demonstrating concurrency without deadlock. Other philosophers, such as P3 and P4, remain in the thinking or waiting state. This diagram reflects how semaphore logic requires philosophers to acquire two forks in a specific order, enabling safe and fair resource sharing.

CONCLUSION

The Dining Philosophers Problem is a classic synchronization issue that illustrates the challenges of resource sharing among concurrent processes. Philosophers need two forks to eat, but they must share them, leading to potential deadlock and starvation if not managed carefully. Semaphores offer a low-level synchronization tool to handle resource access, but they can lead to complex and error-prone solutions.

A simple semaphore-based approach can result in deadlocks, where all philosophers wait indefinitely for forks. Solutions like limiting the number of philosophers allowed to pick forks at once can help, but they require careful handling to avoid starvation. Monitors, on the other hand, provide a higher-level abstraction with built-in synchronization.

By using condition variables and mutual exclusion, monitors ensure that philosophers can wait and be signaled when forks are available, preventing deadlocks and starvation in a more structured way. This makes monitor-based solutions easier to implement and less error-prone.

Both approaches teach important concepts such as deadlock prevention, mutual exclusion, and process coordination. While semaphores offer fine-grained control, monitors simplify synchronization by encapsulating shared resources and operations. Understanding these tools is essential for building efficient, scalable, and safe concurrent systems.

The problem remains a foundational example in operating systems and concurrency theory, emphasizing the importance of careful synchronization design.

FUTURE SCOPE

The Dining Philosophers problem is a classical synchronization problem that provides a foundation for understanding concurrency and resource sharing in operating systems. While the current implementation effectively demonstrates the use of semaphores to manage mutual exclusion and prevent deadlock, several future enhancements can be explored to extend the project's depth, performance, and educational value

The existing solution uses static strategies to avoid deadlocks. Future versions can incorporate more dynamic deadlock avoidance algorithms such as the wait-die or wound-wait schemes. These strategies can improve efficiency and fairness in high-contention environments by making resource allocation decisions based on process priorities or timestamps.

Although deadlock may be avoided, starvation (where a philosopher waits indefinitely to access forks) can still occur. Incorporating starvation prevention mechanisms, such as aging or round-robin request servicing, would make the system more robust and fair to all processes.

In real-world systems, processes often have varying priorities. Enhancing the simulation to allow philosophers with different priorities to access forks based on their importance can help simulate priority-based scheduling policies used in operating systems.

Integrating logging features and metrics such as average waiting time, resource utilization, and throughput would allow for a quantitative assessment of different synchronization strategies. This data could be used to compare the effectiveness of semaphore-based solutions against other methods like monitors or message passing.

The project could be extended to simulate distributed systems, where philosophers represent processes on different nodes and forks represent shared remote resources. This would introduce challenges like network latency and message-based synchronization, further enhancing the realism of the model.

REFERENCES

- Operating Systems: Internals and Design Principles – William Stallings
- Modern Operating Systems – Andrew S. Tanenbaum Python
- Official Documentation – https://docs.python.org
- Pygame Documentation – https://www.pygame.org/docs/
- GeeksforGeeks – Dining Philosophers Problem
- TutorialsPoint – Process Synchronization
- StackOverflow – Python
- Threading Discussions Wikipedia – Semaphore (programming) Wikipedia – Monitor (synchronization)
- GitHub – Python Concurrency Projects
- ResearchGate – Synchronization Techniques
- Real Python – Python Concurrency Tutorials
- W3Schools – Python Threading Medium – OS Concurrency Concepts CS50 Harvard – Operating Systems Lectures
- NPTEL – Operating Systems Course
- Linux Journal – Process Synchronization
- Educative.io – Concurrency in Python
- FreeCodeCamp – Threading Examples Python Module Index
- CodeWithHarry – Python Project
- YouTube – Gaurav Sen on Locks and Concurrency
- IEEE Xplore – Articles on Concurrency
- OS Concepts – Galvin & Silberschatz