VISUALIZATION OF DINING PHILOSOPHERS PROBLEM USING SEMAPHORES AND MONITORS

A Course Based Project Report Submitted in partial fulfilment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY IN

COMPUTER SCIENCE AND ENGINEERING

Submitted by

B.Nanda Kishore	(23071A05E6)
D.Rishi Varma	(23071A05F1)
D.Navadeep	(23071A05F5)
G.Karthik	(23071A05F7)
U.Haradeep Sai	(23071A05K4)

Under the Guidance of

Dr.D N Vasundhara

Assistant Professor, Department of CSE, VNR VJIET

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VALLURUPALLI NAGESWARA RAO VIGNANA JYOTHI INSTITUTE OF ENGINEERING & TECHNOLOGY

An Autonomous, ISO 9001:2015 & QS I-Gauge Diamond Rated Institute, Accredited by NAAC with 'A++' Grade NBA Accreditation for B.Tech. CE,EEE,ME,ECE,CSE,EIE,IT,AME, M.Tech. STRE, PE, AMS, SWE Programmes Approved by AICTE, New Delhi, Affiliated to JNTUH, NIRF (2023) Rank band:101-150 in Engineering Category College with Potential for Excellence by UGC,JNTUH-Recognized Research Centres:CE,EEE,ME,ECE,CSE

Vignana Jyothi Nagar, Pragathi Nagar, Nizampet (S.O.), Hyderabad – 500 090, TS, India.

May, 2025

VALLURUPALLI NAGESWARA RAO VIGNANA JYOTHI

INSTITUTE OF ENGINEERING & TECHNOLOGY

An Autonomous, ISO 9001:2015 & QS I-Gauge Diamond Rated Institute, Accredited by NAAC with 'A++' Grade
NBA Accreditation for B.Tech. CE,EEE,ME,ECE,CSE,EIE,IT,AME, M.Tech. STRE, PE, AMS, SWE Programmes
Approved by AICTE, New Delhi, Affiliated to JNTUH, NIRF (2023) Rank band:101-150 in Engineering Category
College with Potential for Excellence by UGC,JNTUH-Recognized Research Centres:CE,EEE,ME,ECE,CSE

Vignana Jyothi Nagar, Pragathi Nagar, Nizampet (S.O.), Hyderabad – 500 090, TS, India.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the project report entitled "dining philosophers problem using semaphores and monitors" is a bonafide work done under our supervision and is being submitted in the by B.Nanda Kishore(23071A05E6),D.Rishi Varma(23071A05F1),D.Navadeep(23071A05F5), G.Karthik(23071A05F7),U.Haradeep Sai(23071A05K4) in partial fulfilment for the award of the degree of Bachelor of Technology in Computer Science and Engineering, in the branch of CSE of VNRVJIET, Hyderabad during the academic year 2024- 2025.

Certified further that to the best of our knowledge the work presented in this thesis has not been submitted to any other University or Institute for the award of any Degree or Diploma.

Dr.D N Vasundhara Assistant Professor & Internal Guide
CSE Department, VNR VJIET

Dr. V Baby Assoc. Professor & HOD
CSE Department, VNR VJIET

Signature of the External Examiner with date

VALLURUPALLI NAGESWARA RAO VIGNANA JYOTHI
INSTITUTE OF ENGINEERING and TECHNOLOGY

An Autonomous, ISO 9001:2015 & QS I-Gauge Diamond Rated Institute, Accredited by NAAC with 'A++' Grade NBA Accreditation for B.Tech. CE,EEE,ME,ECE,CSE,EIE,IT,AME, M.Tech. STRE, PE, AMS, SWE Programmes Approved by AICTE, New Delhi, Affiliated to JNTUH, NIRF (2023) Rank band:101-150 in Engineering Category College with Potential for Excellence by UGC,JNTUH-Recognized Research Centres:CE,EEE,ME,ECE,CSE

Vignana Jyothi Nagar, Pragathi Nagar, Nizampet (S.O.), Hyderabad – 500 090, TS, India.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DECLARATION

We hereby declare that the course-based project work entitled "**Dining Philosophers Problem Using Semaphores and Monitors**", submitted to the Department of Computer Science and Engineering, Vallurupalli Nageswara Rao Vignana Jyothi Institute of Engineering and Technology (VNRVJIET), Hyderabad, in partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering, is a bonafide record of our own work carried out under the supervision of Dr.D N Vasundhara, Assistant Professor, Department of CSE, VNRVJIET. We also declare that the content of this thesis has not been submitted by us, either in full or in part, for the award of any degree or diploma from any other institution or university previously.

B.Nanda Kishore (23071A05E6) **D.Rishi Varma (23071A05F1)** **D.Navadeep (23071A0F5)** **G.Karthik (23071A05F7)** **U.Haradeep Sai (23071A05K4)**

ACKNOWLEDGEMENT

Firstly, we would like to express our immense gratitude towards our institution VNR Vignana Jyothi Institute of Engineering and Technology, which created a great platform to attain profound technical skills in the field of Computer Science, thereby fulfilling our most cherished goal.

We are very much thankful to our Principal, Dr. Challa Dhanunjaya Naidu and our Head of Department, Dr. V. Baby, for extending their cooperation in doing this project in stipulated time. We extend our heartfelt thanks to our project guide Dr.D.N Vasundhara, for their guidance throughout the course of our project.Last but not least, our appreciable obligation also goes to all the staff members of Computer Science & Engineering department and to our fellow classmates who directly or indirectly helped us.

Mr.B.Nanda Kishore (23071A05E6)
Mr.D.Rishi Varma (23071A05F1)
Mr.D.Navadeep (23071A05F5)
Mr.G.Karthik (23071A05F7)
Mr.U.Haradeep Sai (23071A05K4)

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.

INDEX

.

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.

1. Semaphore-Based Solution

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.

2. Monitor-Based Solution

A monitor is a higher-level synchronization construct that provides a more abstract, safer way to manage concurrency. In a monitor, only one process can execute inside the monitor at any time. It also supports condition variables, which allow processes to wait for certain conditions to be met before proceeding. In this approach, a monitor is used to manage the shared resources (forks) and ensure that philosophers can only pick up both forks if they are available. Key Concepts:

A monitor encapsulates the shared resources and ensures that only one philosopher can enter the critical section at a time. A condition variable (e.g., canEat[i]) is used to wait until both forks are available.

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

Software Requirements

To develop and run the Dining Philosophers Simulation efficiently, the following software components are required: Frontend:

- Python (3.x)
  - The primary programming language used to develop the entire simulation and GUI.
- Tkinter (Python Library)
  - A standard Python library used to build the graphical user interface (GUI) for visualizing the philosophers and their actions (e.g., thinking, eating, waiting for forks).

Development Tools:

- Visual Studio Code (VSCode)
  - A lightweight and powerful code editor with support for Python extensions, debugging, and Git integration.
- PyCharm
  - An integrated development environment (IDE) specifically designed for Python development, which includes useful features like autocompletion, debugging, and project management.
- IDLE (Built-in Python IDE)
  - Python's default integrated development environment for quick script execution and basic debugging.

Browser Support:

- Google Chrome (Recommended)
  - Recommended for testing web-based outputs if the simulation is integrated with a web app in the future.
- Mozilla Firefox
  - An alternative for cross-browser compatibility.
- Microsoft Edge
  - Another browser choice for testing or future web-based versions.

Functional Requirements

These describe the specific functions and features the Dining Philosophers Simulation must offer:

Philosopher Creation –

- The system must initialize philosopher entities around a circular table upon starting the simulation.

Thinking-Hungry-Eating State Cycle –

- Each philosopher must cyclically transition between thinking, hungry, and eating states during the simulation.

Fork Access via Semaphores or Monitors –

- The system must use semaphores or condition variables to synchronize access to shared forks and prevent concurrent access by neighboring philosophers.

Deadlock and Starvation Prevention –

- The simulation must include logic to avoid deadlock and ensure no philosopher starves indefinitely.

Graphical Visualization of States –

- The GUI must display philosophers with color-coded states (e.g., gray for thinking, orange for hungry, green for eating), and forks/spoons visibly connecting them.

Non-Functional Requirements

These define the quality attributes of the system — how well it performs, behaves, and interacts with users.

Performance

- The simulation must respond in real-time, with state changes (thinking, hungry, eating) occurring without noticeable lag.

Usability

- The user interface must be intuitive and easy to use, with clearly labeled buttons and visual indicators for philosopher states and fork usage.

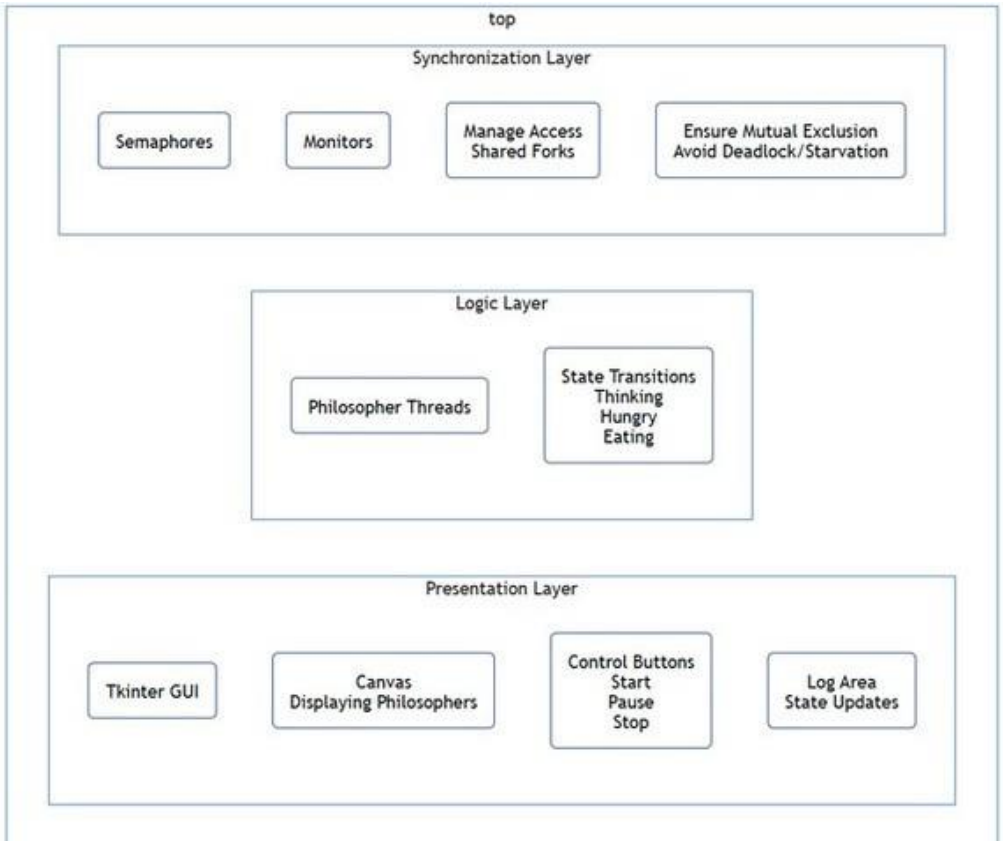
Maintainability

- The code should be modular and well-documented to support easy updates, such as changing the number of philosophers or switching from semaphores to monitors.

Thread Safety

- All interactions involving shared resources (forks) must be thread-safe to prevent race conditions or undefined behavior

ARCHITECTURE DIAGRAM



The architecture of the Dining Philosophers Simulation is divided into three layers: the Presentation Layer, Logic Layer, and Synchronization Layer. The Presentation Layer, built using Tkinter, provides the graphical interface including a canvas to visualize philosophers and forks, control buttons (Start, Pause, Stop), and a text log to display philosopher actions in real-time. The Logic Layer handles the simulation flow by representing each philosopher as a separate thread that transitions between thinking, hungry, and eating states. The Synchronization Layer manages access to shared resources (forks) using semaphores or condition variables to ensure mutual exclusion, prevent deadlock, and avoid starvation. These layers interact seamlessly to provide a thread-safe, responsive, and educational simulation of concurrency in operating systems.

MODULES

1.User Interface Module

Responsibilities:

- Handles the graphical representation of philosophers and forks.
- Displays philosopher states using colors and visual elements.
- Provides control buttons (Start, Pause, Stop) and a log area.
- Technologies Used:
- Pygame (or Tkinter, depending on your version)
-

2. Philosopher Module

Responsibilities:

- Defines each philosopher's behavior and state transitions (Thinking → Hungry → Eating).
- Runs each philosopher as a separate thread to simulate concurrency.
- Updates state visually and logs actions.

3.Semaphore Module

Responsibilities:

- Manages fork acquisition using semaphores to ensure mutual exclusion. Prevents two
- adjacent philosophers from accessing the same fork at the same time. Handles
- synchronization logic to avoid resource conflicts.

4. Monitor Module

Responsibilities:

- Uses monitors (condition variables and locks) to manage access to
- forks.
- Ensures thread-safe fork handling and prevents deadlock or starvation.
- Coordinates philosopher access based on availability of resources.

5. Controller Module

Responsibilities:

- Acts as the interface between user input and internal simulation logic.
- Starts, pauses, and stops the simulation based on user commands.
- Coordinates communication among UI, Philosopher, and Synchronization modules.

MODULE WISE DESCRIPTION

1. UI Module

This module is responsible for rendering the simulation environment. It displays philosophers, forks, and their current state (thinking, hungry, eating) using graphical elements and color codes. The simulation uses Pygame's rendering system and updates in real-time.

* It handles all visual feedback and user interface rendering.

2. Philosopher Module

Each philosopher is represented as an object with properties like ID and state. This module governs the transitions between states and calls synchronization functions to request or release forks.

* It simulates individual philosopher behavior using threading and state logic.

3. Semaphore Module

Implements classical semaphore-based synchronization. It ensures mutual exclusion by acquiring both forks using semaphores. It includes logic to prevent deadlock using resource hierarchy or odd/even philosopher strategy.

* It provides low-level locking mechanisms to control fork access.

4. Monitor Module

Implements a higher-level synchronization abstraction using Python's Condition object. It ensures philosophers wait when resources aren't available and resumes them appropriately, preventing race conditions.

* It uses monitors to manage safe access and state coordination.

5. Controller Module

This acts as the coordinator. It manages the simulation state, responds to user interactions (such as switching between monitor/semaphore), and ensures consistent updates in the visual interface.

* It links the UI with logic modules to control simulation flow.

IMPLEMENTATION USING SEMAPHORES

```
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, 2))
    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)
```

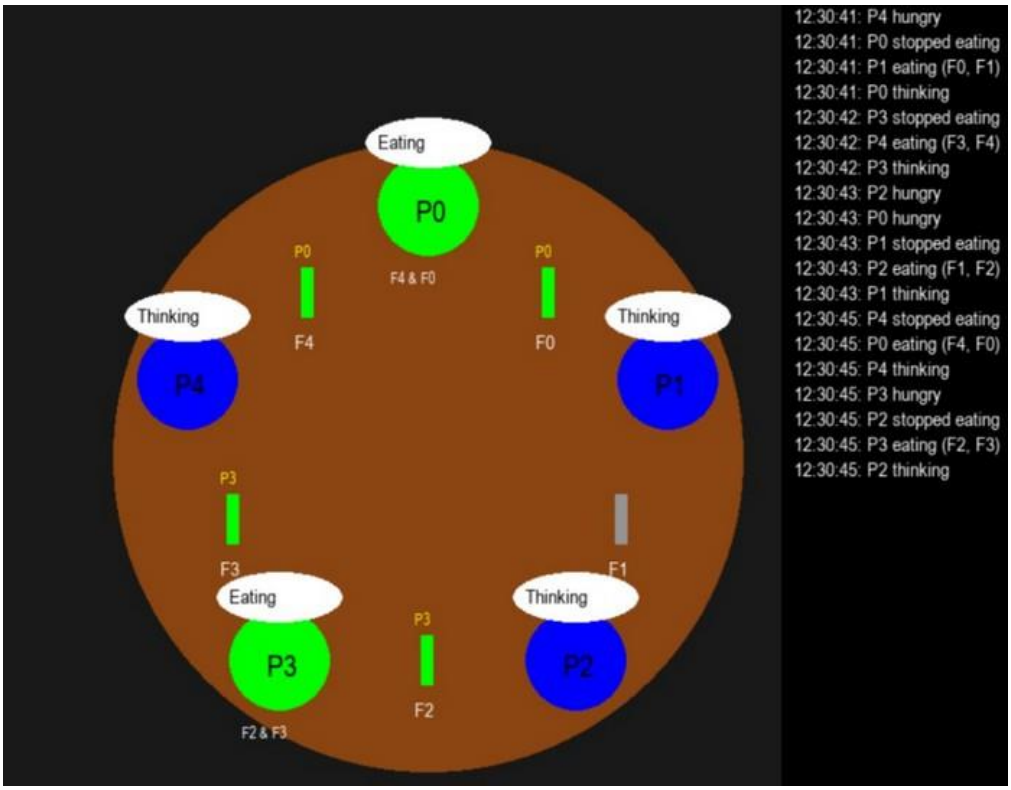
```
def pick_forks(self, left_fork_index, right_fork_index):
    if self.stop_event.is_set():
        return
    # Draw fork lines when philosopher picks up forks, show which fork was picked first
    color = "blue" if self.index % 2 == 0 else "green"  # Blue for even index, Green for odd
    fork_line_left = 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()
```

```
    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")
```

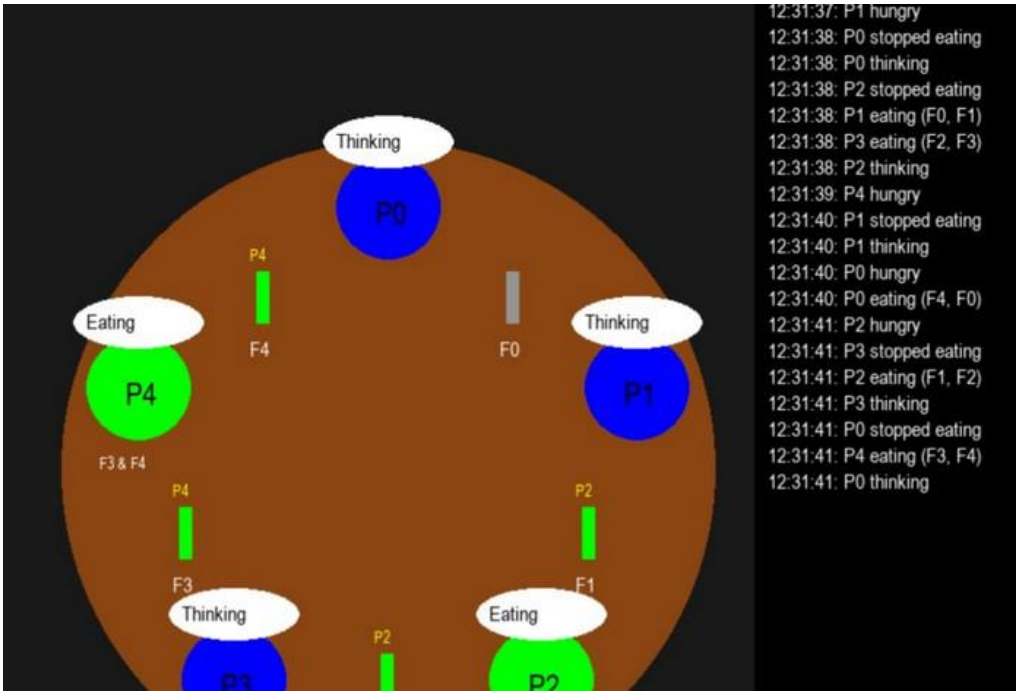
```
  spoon_radius = 120
  self.spoon_positions = []
  for i in range(num_philosophers):
      angle = angle_step * i
      x = center[0] + spoon_radius * cos(radians(angle))
      y = center[1] + spoon_radius * sin(radians(angle))
      self.spoon_positions.append((x, y))
      self.canvas.create_line(center[0], center[1], x, y, width=5, fill="gray")
      # 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()
```

RESULTS



This image illustrates the initial state of the dining philosophers problem using a semaphore-based synchronization mechanism. All five philosophers are positioned evenly around a circular table, and each one is in the thinking state, indicated by their light grey head color. Forks are positioned between the philosophers, marked numerically for clarity. The background shows no active fork usage, and no philosopher is hungry or eating yet. This clean setup provides a base state to observe how semaphore coordination will later handle fork acquisition. It emphasizes that the system begins in a deadlock-free, non- contended state where no semaphore has been claimed yet.



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.

IMPLEMENTATION USING MONITORS

```
import pygame
import math
import threading
import time
import random
from collections import deque

pygame.init()

# Constants
WIDTH, HEIGHT = 1200, 800
CENTER = (WIDTH // 2, HEIGHT // 2)
RADIUS = 200
NUM_PHILOSOPHERS = 5
TIMEOUT = 5  # seconds for fork pickup timeout

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
FORK_AVAILABLE = (150, 150, 150)
FORK_USED = (0, 255, 0)
FORK_TEXT_COLOR = (255, 255, 0)

# Philosopher states
THINKING = 0
HUNGRY = 1
EATING = 2
DEADLOCK = 3
STATE_NAMES = {
    THINKING: "Thinking",
    HUNGRY: "Hungry",
    EATING: "Eating",
    DEADLOCK: "Deadlock!"
}
STATE_COLORS = {
    THINKING: (0, 0, 255),
    HUNGRY: (255, 165, 0),
    EATING: (0, 255, 0),
    DEADLOCK: (255, 0, 0)
}

screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Dining Philosophers Problem")

font = pygame.font.SysFont('Arial', 16)
big_font = pygame.font.SysFont('Arial', 24)
small_font = pygame.font.SysFont('Arial', 12)

class Monitor:
    def _init_(self):
        self.state = [THINKING] * NUM_PHILOSOPHERS
        self.forks_in_use = [None] * NUM_PHILOSOPHERS
        self.condition = threading.Condition()
        self.log = deque(maxlen=20)
        self.log_lock = threading.Lock()
        self.running = True
```

```
  def log_event(self, message):
      with self.log_lock:
          self.log.append(f"{time.strftime('%H:%M:%S')}: {message}"

  def test(self, i):
      left = (i + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS
      right = i
      if (self.state[i] == HUNGRY and
          self.forks_in_use[left] is None and
          self.forks_in_use[right] is None):
          self.state[i] = EATING
          self.forks_in_use[left] = i
          self.forks_in_use[right] = i
          self.condition.notify_all()
          self.log_event(f"P{i} eating (F{left}, F{right})")

  def pickup_forks(self, i):
      with self.condition:
          if not self.running:
              return False
          self.state[i] = HUNGRY
          self.log_event(f"P{i} hungry")
          self.test(i)
          start_time = time.time()
          while self.state[i] != EATING and self.running:
              remaining = TIMEOUT - (time.time() - start_time)
              if remaining <= 0:
                  self.state[i] = DEADLOCK
                  self.log_event(f"P{i} timed out!")
                  return False
              self.condition.wait(remaining)
          return self.running and self.state[i] == EATING
```

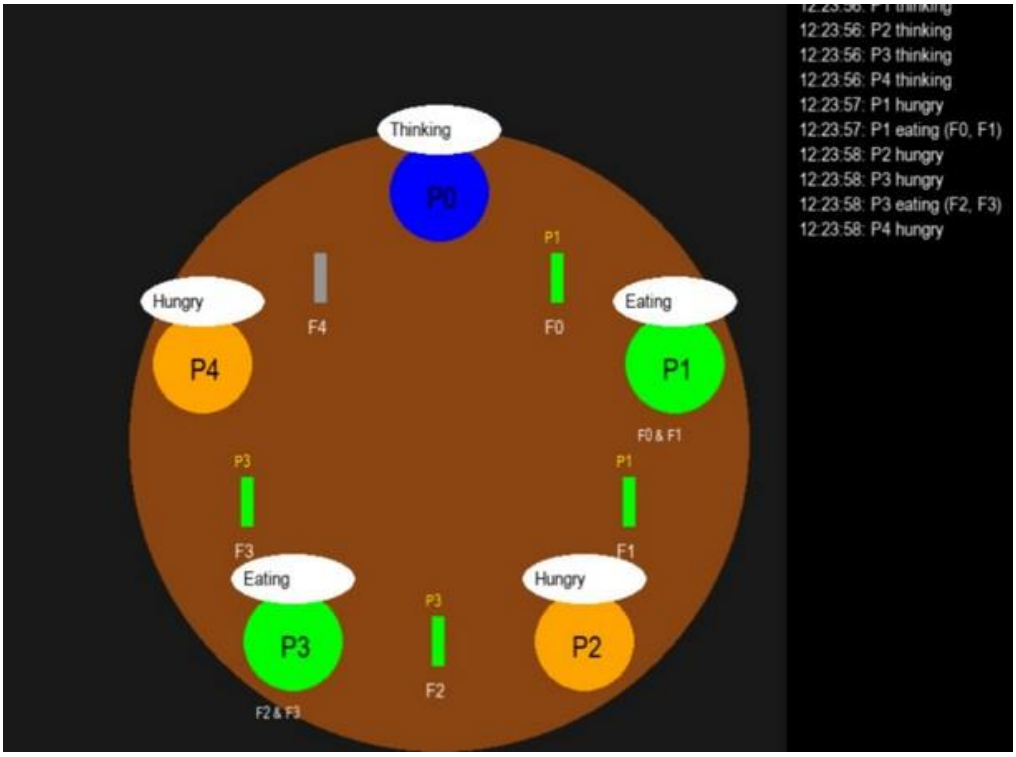
```
    def putdown_forks(self, i):
        with self.condition:
            left = (i + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS
            right = i
            self.state[i] = THINKING
            self.forks_in_use[left] = None
            self.forks_in_use[right] = None
            self.log_event(f"P{i} stopped eating")
            self.test((i - 1 + NUM_PHILOSOPHERS) % NUM_PHILOSOPHERS)
            self.test((i + 1) % NUM_PHILOSOPHERS)

    def stop(self):
        with self.condition:
            self.running = False
            self.condition.notify_all()

def philosopher(id, monitor):
    while monitor.running:
        monitor.log_event(f"P{id} thinking")
        time.sleep(random.uniform(1, 3))
        if not monitor.running:
            break
        if monitor.pickup_forks(id):
            time.sleep(random.uniform(1, 3))
            monitor.putdown_forks(id)

def draw_scene(monitor, button_rect):
    screen.fill((25, 25, 25))

    # Table and philosophers
    pygame.draw.circle(screen, (139, 69, 19), CENTER, RADIUS + 50)
    positions = []
    for i in range(NUM_PHILOSOPHERS):
        angle = 2 * math.pi * i / NUM_PHILOSOPHERS - math.pi / 2
        x = CENTER[0] + RADIUS * math.cos(angle)
        y = CENTER[1] + RADIUS * math.sin(angle)
        positions.append((int(x), int(y)))
```

```
    # Forks
    for i in range(NUM_PHILOSOPHERS):
        p1 = positions[i]
        p2 = positions[(i + 1) % NUM_PHILOSOPHERS]
        fx = (p1[0] + p2[0]) // 2
        fy = (p1[1] + p2[1]) // 2
        used_by = monitor.forks_in_use[i]
        color = FORK_USED if used_by is not None else FORK_AVAILABLE
        pygame.draw.rect(screen, color, (fx - 5, fy - 20, 10, 40))
        fork_label = font.render(f"F{i}", True, WHITE)
        screen.blit(fork_label, (fx - 10, fy + 30))
        if used_by is not None:
            usage = small_font.render(f"P{used_by}", True, FORK_TEXT_COLOR)
            screen.blit(usage, (fx - 10, fy - 40))

    # Philosophers
    for i, pos in enumerate(positions):
        state = monitor.state[i]
        color = STATE_COLORS[state]
        pygame.draw.circle(screen, color, pos, 40)
        label = big_font.render(f"P{i}", True, BLACK)
        screen.blit(label, (pos[0] - 10, pos[1] - 10))
        pygame.draw.ellipse(screen, WHITE, (pos[0] - 50, pos[1] - 70, 100, 40))
        state_text = font.render(STATE_NAMES[state], True, BLACK)
        screen.blit(state_text, (pos[0] - 40, pos[1] - 60))
        if state == EATING:
            left = (i + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS
            right = i
            fork_text = small_font.render(f"F{left} & F{right}", True, WHITE)
            screen.blit(fork_text, (pos[0] - 30, pos[1] + 50))
```

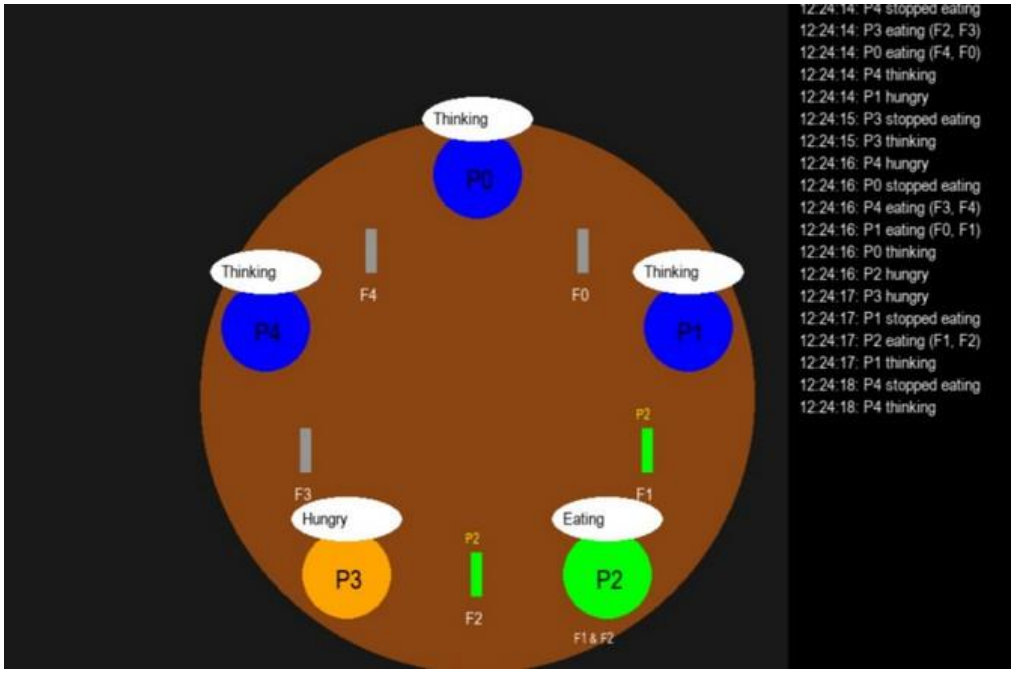
```
    # Log Panel (Right side)
    panel_x = WIDTH // 2 + RADIUS + 80
    pygame.draw.rect(screen, BLACK, (panel_x, 10, 360, HEIGHT - 120))
    log_y = 20
    with monitor.log_lock:
        for msg in list(monitor.log)[-20:]:
            log_text = font.render(msg, True, WHITE)
            screen.blit(log_text, (panel_x + 10, log_y))
            log_y += 20

    # Control Button at bottom center
    btn_text = "Stop" if monitor.running else "Start"
    btn_label = big_font.render(btn_text, True, WHITE)
    button_rect = btn_label.get_rect(center=(WIDTH // 2, HEIGHT - 50))
    pygame.draw.rect(screen, (0, 100, 200), button_rect.inflate(20, 10))
    screen.blit(btn_label, button_rect)

    pygame.display.flip()
    return button_rect

def main():
    monitor = Monitor()
    philosophers = []
    for i in range(NUM_PHILOSOPHERS):
        t = threading.Thread(target=philosopher, args=(i, monitor))
        t.start()
        philosophers.append(t)

    clock = pygame.time.Clock()
    button_rect = None
```

```
    try:
        while True:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    raise KeyboardInterrupt
                if event.type == pygame.MOUSEBUTTONDOWN and button_rect:
                    if button_rect.collidepoint(event.pos):
                        if monitor.running:
    else:
        monitor.running = True
        for i in range(NUM_PHILOSOPHERS):
            if monitor.state[i] == DEADLOCK:
                monitor.state[i] = THINKING
        philosophers = []
        for i in range(NUM_PHILOSOPHERS):
            t = threading.Thread(target=philosopher, args=(i, monitor))
            t.start()
            philosophers.append(t)
```

RESULTS



This image shows the monitor-based implementation of the problem, likely after pressing the "Stop" button or at the beginning of a new simulation run. All philosophers are in the thinking state (grey heads), and the system appears idle. Notably, fork labels are now denoted with the prefix ‘F’ (F1 to F5), clearly marking each fork’s position relative to the table and philosophers. The scene is visually balanced, and no fork acquisition lines are visible. This idle state indicates that no thread is holding a lock or waiting on condition variables, underscoring the monitor’s advantage in allowing safe reset and clean state transitions without thread contention.



This frame presents the monitor-based solution under active contention, showing an efficient and fair access model. Philosopher 5 and Philosopher 2 are eating (green heads), having successfully acquired both adjacent forks. Their fork usage is marked by blue lines. Meanwhile, Philosopher 4 is in the hungry state, represented with an orange head, signaling a pending condition that will be handled when forks become available. The monitor model internally uses a condition variable and mutual exclusion lock to ensure that no two neighboring philosophers eat at the same time, and that state changes (thinking, hungry, eating) are coordinated without race conditions. This image highlights how monitor synchronization ensures deadlock-free operation with smoother state transitions, while also being visually intuitive with dynamic updates.

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.

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