كلية الحاسبات والذكاء الإصطناعي
Faculty of Computers & Artificial Intelligence

جامعة حلوان
HELWAN UNIVERSITY

Cybersecurity Diploma
Operating System

Project 3
Case # (3) - Dining Philosopher's problem

prepared by

Ahmed Mohamed Maged
Wesam Ahmed Hassan
Mohamed Tarek Taha Ahmed
Mahmoud Ahmed AbdelAziz
Mahmoud Osama Ibrahim

Supervised by

Dr. Ahmed Hesham

Documentation

1. Dining Philosopher’s problem:

This problem states that there are 5 philosophers sitting around a circular table with one chopstick placed between each pair of philosophers. The philosopher will be able to eat if he can pick up two chopsticks that are adjacent to the philosopher. This problem deals with the allocation of limited resources.

This document explains the implementation of the Dining Philosophers Problem using Python. It ensures synchronization between threads while avoiding deadlocks and starvation

---

2. Pseudocode:

```
1. Define Chopstick class with:
   - Initialize ID.
   - Initialize lock.

2. Define Philosopher class that extends Thread:
   - Initialize ID, left chopstick, right chopstick, and priority lock.
   - Define run method:
     While True:
        1. Think
        2. Request priority lock
        3. Pick up left chopstick
        4. Pick up right chopstick
        5. Eat.
        6. Release chopsticks.

3. Define DiningPhilosophers class:
   - Initialize list of chopsticks and philosophers.
   - Define start_dining method to start philosopher threads.

4. In main:
   - Create DiningPhilosophers instance with number of philosophers
   - Call start_dining method
```

3. Solution Explanation:

- **The solution implements the Dining Philosophers Problem using multithreading in Python**.
- **Each philosopher is represented as a thread that alternates between thinking and eating.**
- **Chopsticks are resources, and each philosopher needs two (the left and right chopsticks) to eat.**
- **To prevent deadlock, a priority lock is used. Philosophers must acquire this lock before trying to pick up chopsticks, ensuring that no philosopher can hold one chopstick while waiting for another.**
- **The code is structured using OOP principles, with clear classes for Chopstick, Philosopher, and DiningPhilosophers, making it modular and easy to understand.**

---

4. Deadlock Example and Solution

- **Deadlock Scenario:**
  - **If each philosopher picks up their left chopstick first and waits for the right chopstick, a deadlock occurs where no philosopher can proceed.**

- **Solution:**
  - **By using a priority lock, philosophers must acquire this lock before picking up chopsticks, preventing the situation where they hold one chopstick and wait for another.**

---

5. Starvation Example and Solution

- **Starvation Scenario:**
  - **If a philosopher is repeatedly preempted while trying to eat (e.g., due to high contention), they may never get both chopsticks.**

- **Solution:**
  - **The starvation monitor tracks mealtimes. If a philosopher hasn’t eaten within a set timeframe, a warning is printed, prompting a review of the system's resource allocation.**

Python Code

Here's a Python implementation of the monitor-based Dining Philosophers problem using the threading module:

1. Chopstick Class

The Chopstick class represents a shared resource (chopstick) between philosophers .

- **Attributes:**
  - id: Unique identifier for chopsticks.
  - lock: Lock to prevent multiple philosophers from using the same chopstick simultaneously.

- **Explanation:**
  - Each Chopstick is assigned an ID for easy identification.
  - A Lock is associated with each chopstick to prevent more than one philosopher from using the same chopstick at the same time.

---

2. Starvation Monitor Class

This class defines the Starvation Monitor class to track philosophers' hunger state.

- **Attributes:**
  - philosophers: List of philosophers.
  - timeout: Maximum allowed time without eating.
  - last_meal_times: Store the last mealtime for each philosopher.
  - monitor_thread: Separate thread to monitor starvation.

- **Explanation:**
  - The StarvationMonitor tracks how long each philosopher has gone without eating.
  - It uses a dictionary to store the last meal time for each philosopher.
  - A separate thread (monitor_thread) continuously checks for any philosopher who exceeds the starvation timeout.

---

3. `Philosopher` Class

The Philosopher class models each philosopher as a thread in the simulation.

- **Attributes:**
  - `id`: Unique identifier for the philosopher.
  - `left_chopstick`: The chopstick on the left.
  - `right_chopstick`: The chopstick on the right.
  - `priority_lock`: Lock to manage priority and avoid conflicts.
  - `starvation_monitor`: Reference to the starvation monitor.

- **Explanation:**
  - Each Philosopher is initialized with an ID and references to their left and right chopsticks.
  - A shared priority_lock ensures that only one philosopher attempts to pick up chopsticks at a time..
  - The starvation_monitor updates meal times when philosophers eat.
  - Each philosopher alternates between thinking and eating in a continuous loop.
  - The priority_lock ensures that philosophers request chopsticks in an organized manner.
  - Locks for the left and right chopsticks are acquired sequentially to simulate eating.
  - Once the philosopher eats, the last meal time is updated in the StarvationMonitor.

---

4. `dining_philosophers` Main Simulation Function

```
def dining_philosophers(num_philosophers):
    chopsticks = [Chopstick(i) for i in range(num_philosophers)]
    priority_lock = threading.Lock()

    philosophers = [
        Philosopher(i, chopsticks[i], chopsticks[(i + 1) %
num_philosophers], priority_lock, None)
        for i in range(num_philosophers)
    ]
```

- This function creates a list of chopsticks equal to the number of philosophers.
- Create philosophers, assigning each one a left and right chopstick.
- Use a shared priority_lock to avoid conflicts when philosophers attempt to pick up chopsticks.

```
    starvation_monitor = StarvationMonitor(philosophers)
    for philosopher in philosophers:
        philosopher.starvation_monitor = starvation_monitor
```

- Initialize the StarvationMonitor to track philosophers' eating times.
- Assign the monitor to each philosopher.

```
        starvation_monitor.monitor_thread.start()

    for philosopher in philosophers:
        philosopher.start()  # Start philosopher threads
```

- Start the starvation monitor thread.
- Start each philosopher thread, allowing them to run independently.

```
    try:
        while True:
            time.sleep(1)  # Keep the program running
    except KeyboardInterrupt:
        print("\nStopping simulation...")
        starvation_monitor.stop()  # Stop monitor
        starvation_monitor.monitor_thread.join()
```

- The main thread keeps the program running until interrupted (e.g., Ctrl+C)..
- When interrupted, the simulation stops gracefully by stopping the starvation monitor.

---

5. Program Entry Point

```
    try:
        while True:
            time.sleep(1)  # Keep the program running
    except KeyboardInterrupt:
        print("\nStopping simulation...")
        starvation_monitor.stop()  # Stop monitor
        starvation_monitor.monitor_thread.join()

dining_philosophers(5)
```

Run the simulation with 5 philosophers by calling the dining_philosophers function.

6. Overall Summary

This program simulates the Dining Philosophers Problem:

- **Philosophers alternate between thinking and eating.**
- **Chopsticks (resources) are shared and accessed using locks.**
- **A Starvation Monitor ensures no philosopher goes too long without eating.**

---

7. Key Features

- **Deadlock prevention with priority locking**.
- **Starvation detection through a dedicated monitor thread.**
- **Multi-threaded simulation of philosophers' actions.**
- **Efficient resource management using locks for chopsticks.**
- **Dynamic scalability for any number of philosophers.**
- **Graceful program shutdown with manual interruption.**