Project Title:

Dining Philosophers Problem

Submitted by:

Kashif Muneer	**2021-CE-34**
Mehmood Ul Haq	**2021-CE-35**
Shahzaib	**2021-CE-41**
Talal Muzammal	**2021-CE-47**

Submitted to:

Darakhshan Abdul Ghaffar

Course:

CMPE-331L: Operating Systems A

Semester:

05

Department of Computer Engineering

University of Engineering and Technology, Lahore

Philosopher Dining Hall Problem

Classical synchronization in operating systems refers to the set of mechanisms and techniques used to coordinate and control the execution of multiple threads or processes to ensure correct and predictable behaviour in a concurrent computing environment. Concurrency arises when multiple tasks or processes run simultaneously, and synchronization is crucial to manage shared resources and avoid conflicts. The classical synchronization problem is often associated with the challenges posed by concurrent access to shared data.

Key concepts related to classical synchronization include:

Mutual Exclusion:

Ensuring that only one thread or process accesses a critical section (a segment of code that modifies shared data) at a time. This prevents interference and data corruption.

Critical Sections:

The portions of code that access shared resources and need to be executed in a mutually exclusive manner.

Semaphore:

A synchronization primitive that allows multiple threads to coordinate access to shared resources. Semaphores maintain a count to control access.

Mutex (Mutual Exclusion Lock):

A type of synchronization mechanism that ensures mutual exclusion. A thread that acquires a mutex gains exclusive access to a critical section.

Deadlock:

A situation where two or more processes are unable to proceed because each is waiting for the other to release a resource. Deadlocks can occur in systems with improper synchronization.

Starvation:

A condition where a thread or process is perpetually denied access to a resource it needs due to improper synchronization policies.

Condition Variables:

Mechanisms that allow threads to wait for a specific condition to be met before proceeding. They are often used in conjunction with mutexes.

Producer-Consumer Problem:

A classical synchronization problem where one or more producer threads generate data, and one or more consumer threads consume the data. Proper synchronization ensures that producers and consumers operate correctly and do not interfere with each other.

Readers-Writers Problem:

A synchronization problem involving multiple readers and writers accessing shared data. Synchronization mechanisms prevent conflicts between readers and writers.

Effective classical synchronization is essential for the correct and efficient functioning of concurrent programs, ensuring data integrity and preventing race conditions. Modern operating systems provide a variety of synchronization primitives and tools to address these challenges.

Deadlock is a situation in which two or more processes are unable to proceed because each is waiting for the other to release a resource. In other words, it's a state where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

In the context of the **Dining Philosophers Problem**, deadlock can occur when each philosopher holds one fork and waits indefinitely for the other fork to become available. Let's break down how deadlock can happen in this scenario:

1. **Circular Waiting:**
    - The dining philosopher's problem involves a circular arrangement of philosophers, each represented as a node in a circle, and each node has two adjacent forks.
    - For deadlock to occur, each philosopher must pick up one fork and then wait for the other fork, creating a circular waiting dependency.
2. **Symmetric Action:**
    - Philosophers perform symmetric actions: picking up the left fork and the right fork.

- If every philosopher picks up one fork and then waits for the adjacent fork (held by the next philosopher), a circular waiting pattern emerges.

3. **No Release of Resources:**
    - In a deadlock, the crucial condition is that no philosopher releases any of the forks they are holding.
    - As each philosopher holds one fork and waits for the other, no progress is possible, resulting in a deadlock.

In summary, deadlock in the dining philosopher's problem arises when all philosophers cyclically wait for the fork to their right (or left), and none of them releases the fork they are holding. This can lead to a situation where no philosopher can make progress, and the system remains in a blocked state.

To avoid deadlock in the dining philosopher's problem, various synchronization mechanisms can be employed, such as using semaphores or enforcing rules that prevent circular waiting. These solutions ensure that philosophers can acquire all necessary resources (forks) without forming a circular dependency that leads to a deadlock.

Why we have used semaphores in our philosopher dining problem?

Semaphores are a good solution to the dining philosopher's problem because they provide a mechanism for controlling access to shared resources and help prevent potential issues such as deadlock. In the context of the dining philosopher's problem, semaphores are used to manage access to the forks, ensuring that philosophers can pick up and release forks in a coordinated manner.

Here are the key reasons why semaphores are a suitable solution for the dining philosopher's problem:

1. **Mutex Functionality:**
    - Semaphores can be used as binary semaphores (with a maximum count of 1) to provide mutual exclusion.
    - This ensures that only one philosopher can access a critical section of code (e.g., picking up or releasing forks) at a time, preventing race conditions.
2. **Controlling Resource Access:**

- Semaphores allow for the controlled access to shared resources, such as forks in the dining philosopher's problem.
- By using a semaphore, you can limit the number of philosophers who can hold forks simultaneously, addressing the potential for deadlock.

3. **Avoiding Deadlock:**
    - Deadlock can occur when each philosopher holds one fork and waits indefinitely for the other. Semaphores can help prevent this situation.
    - By limiting the number of philosophers who can pick up forks simultaneously (using a semaphore), the system avoids scenarios where all philosophers attempt to pick up a single fork simultaneously.
4. **Signalling and Coordination:**
    - Semaphores can be combined with condition variables or other signalling mechanisms to coordinate the actions of philosophers.
    - In the provided code, condition variables (**cond**) are used to signal philosophers when it's appropriate to proceed (e.g., when a fork is available).
5. **Resource Releasing:**
    - Semaphores facilitate the release of resources in a controlled manner.
    - In the dining philosopher's problem, when a philosopher puts down a fork, it releases the semaphore, allowing another philosopher to potentially pick it up.

While semaphores provide a solid foundation for solving the dining philosophers' problem, it's important to note that the specific implementation details may vary. More advanced synchronization techniques, such as using monitors or combining semaphores with mutexes and condition variables, can be explored to address additional challenges and ensure robustness in more complex scenarios.

Application of dining philosopher problem in Operating Systems:

The dining philosopher’s problem is a classic synchronization problem that highlights concurrent programming and resource allocation challenges. Here's how the code relates to operating systems:

1. **Concurrency and Resource Allocation:**
    - In operating systems, multiple processes or threads often compete for shared resources.
    - The dining philosopher’s problem represents a scenario where multiple philosophers (threads/processes) compete for access to forks (resources).
2. **Semaphore Usage:**
    - The use of semaphores in the code is a common synchronization technique. Semaphores help control access to shared resources and avoid race conditions.
3. **Deadlock Avoidance:**
    - The code uses a semaphore to limit the number of philosophers that can eat simultaneously, helping prevent deadlock.
    - Deadlock in an operating system occurs when processes are blocked waiting for resources held by other processes, creating a cycle.

Real-World Application of Philosopher Dining Problem:

While the dining philosophers' problem is theoretical, its principles and solutions can be applied to real-world scenarios involving resource management and concurrency. Here are some real-world applications:

1. **Database Management:**
    - In a database system, multiple transactions may compete for access to shared data.
    - The principles of avoiding deadlock and efficient resource allocation apply to ensure that transactions can proceed without conflicts.
2. **Multithreaded Programming:**
    - In applications with multiple threads, such as web servers, resource contention may arise.

- Proper synchronization techniques, inspired by dining philosophers' solutions, can be employed to manage access to shared data structures.

3. **Network Resource Allocation:**
    - In networking, multiple devices may compete to access a shared network resource, such as a communication channel.
    - Synchronization techniques can be used to ensure fair and efficient access to network resources.
4. **Task Scheduling:**
    - In an operating system's task scheduler, multiple processes may compete for CPU time.
    - Techniques to prevent deadlock and ensure fair scheduling are crucial for efficient system operation.

While the dining philosopher's problem itself is a simplified and academic scenario, the concepts it explores are relevant to a variety of real-world situations where resource contention and concurrency are common challenges. Real-world applications often involve more complex scenarios and require sophisticated synchronization mechanisms to ensure robust and efficient operation.

Code Explanation:

Let's dive deeper into the three libraries imported at the beginning of your code: **threading**, **time**, and **random**.

1. threading:

The **threading** module provides a way to work with threads in Python. Threads are lightweight sub-processes that run concurrently, allowing different parts of a program to execute independently. Key components of the **threading** module include:

- **Thread class:**
    - The **Thread** class is used to create and manage threads.
    - Threads are created by instantiating objects of this class and passing a target function that the thread will execute.

- **Locks and Conditions:**
    - The module provides synchronization primitives like locks (**Lock** class) and conditions (**Condition** class).
    - Locks are used to prevent multiple threads from accessing shared resources simultaneously, avoiding race conditions.
    - Conditions allow threads to coordinate their execution and communicate with each other.
- **Thread Safety:**
    - The module supports thread safety through various mechanisms, allowing developers to write concurrent programs with shared resources in a safe way.

2. time:

The **time** module deals with various time-related functions. Some of the commonly used functions include:

- **time.sleep(seconds)**:
    - Pauses the execution of the current thread for the specified number of seconds.
    - Useful for introducing delays or controlling the timing of actions in a program.
- **Timestamps and Clocks:**
    - The module provides functions for working with timestamps and measuring time intervals.
- **Time Formatting:**
    - Functions for formatting and parsing time in different representations.

3. random:

The **random** module is used for generating random numbers and performing random-related operations. Key features include:

- **Random Number Generation:**
    - Functions like **random.randint(a, b)** generate a random integer between **a** and **b** (inclusive).

    - Useful for introducing variability or randomness into the behavior of a program.
- **Random Sequences:**
    - Functions for shuffling sequences (**random.shuffle()**) or selecting random elements from sequences (**random.choice()**).
- **Seed Control:**
    - The **random.seed()** function allows you to initialize the random number generator with a specific seed, ensuring reproducibility.

Use in the Dining Philosophers Program:

In the context of your dining philosophers program:

- **threading** is used to create and manage threads for each philosopher.
- **time** is likely used for introducing delays between actions or for simulating the passage of time.
- **random** is likely used for generating random durations for thinking, eating, and other activities, adding variability to the simulation.

Together, these libraries provide the necessary tools for implementing concurrency, synchronization, and simulation of time-related activities in your program.

1. **NUMPHIL = 5:**
    - Constant representing the number of philosophers in the dining philosopher's problem (5 philosophers in this case).
2. **LEFT = lambda i: (i - 1 + NUMPHIL) % 5 and RIGHT = lambda i: (i + 1) % 5:**
    - Lambda functions defining circular indices for the left and right neighbors of a philosopher.
    - Ensures that philosophers are seated in a circular arrangement, and the indices wrap around appropriately.
3. **Enumeration for Philosopher States:**
    - **THINKING**, **HUNGRY**, and **EATING** are assigned values 0, 1, and 2, respectively.

- These constants represent the possible states of a philosopher: thinking, hungry, and eating.
- Used for readability in the program to check and update the state of each philosopher.

4. **mutex = threading.Lock():**

This line creates a mutex, short for "mutual exclusion," using the Lock class from the threading module. The mutex is a synchronization primitive that ensures exclusive access to a shared resource. In this context, it is likely used to protect critical sections of the code where multiple threads might access shared variables concurrently. The mutex will help prevent race conditions by allowing only one philosopher to enter the critical section at a time.

5. **forks = [threading.Semaphore(1) for _ in range(NUMPHIL)]:**

This line creates a list of threading.Semaphore objects called forks. A semaphore is a synchronization primitive that can be used to control access to a resource with limited capacity. In this case, each semaphore in the list represents a fork, and the value 1 indicates that the fork is available.

- The list comprehension [threading.Semaphore(1) for _ in range(NUMPHIL)] creates a list of NUMPHIL semaphores, each initialized with a capacity of 1.
- Each semaphore can be thought of as representing a fork on the dining table.

These semaphores will be used to coordinate access to the forks, ensuring that a philosopher can pick up a fork only if it is available (semaphore value is 1). If a philosopher is holding a fork, the semaphore value is decreased, and when the philosopher puts the fork back, the semaphore value is increased.