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

Date of Submission:

27-Dec-2023

Department of Computer Engineering

University of Engineering and Technology, Lahore

Table of Contents

Abstract:

This study shows how to avoid deadlock condition in dining philosophers' problem. It is the unbearable situation of concurrent systems. It shows as being in a circular waiting state. Originally, most people wear ideas simply supported by the hardware. For example, the user and user interrupt routines that have been installed by the hardware. In 1967 Dijkstra introduced the concept wearer an integer variable, which counts the number of processes which are active or the number which are inactive. The provided Python code implements the dining philosopher's problem using threading and synchronization mechanisms. The dining philosopher's problem is a classic example in computer science, illustrating challenges related to resource allocation and deadlock prevention in a concurrent system. In this scenario, a fixed number of philosophers (defined by NUMPHIL) engage in a cyclic sequence of thinking, eating, and waiting for forks.

Each philosopher is modeled as a separate thread, and their actions are coordinated using locks and conditions. The philosophers alternate between thinking, attempting to pick up forks to eat, eating, and returning the forks. The state of each philosopher is tracked to prevent conflicts and ensure that no two adjacent philosophers are eating simultaneously. The code employs a lock to ensure mutual exclusion and conditions to manage the synchronization between philosophers. Also, this paper describes an efficient distributed deadlock avoidance scheme which uses lock and release method to avoid the race condition between other threads in the chain.

Introduction

An operating system is a program which links the user and the computer system. This operating system has to have a way of controlling resources. When designing the operating system, concurrency is a common foundation. Simultaneous processes are when the processes run at the same time. This is the multitasking operating system. As for concurrent processes, they can be completely independent of each other, or they can interact with each other. Only properly-controlled processes that must synchronize to interact are allowed. Nevertheless, among the concurrent processes that interact with one another, there are problems to be overcome such as deadlock and synchronization.

Problem statement

One such classic problem that can show what the problem is the Dining Philosophers Problem. The Dining Philosopher's Problem is something like this: there are five philosophers who want to eat. Five chopsticks had been reserved on the table. Philosophers are really hungry. Two chopsticks should be in the right and left hands. But sometimes only one course uses chopsticks. Therefore, if there are philosophers who take two chopsticks, then there are philosophers who have to wait for the chopsticks to be put back. There lies within this problem the potential for deadlock, which occurs when two or more processes cannot proceed.

Solution to the problem

The dining philosopher's problem is a typical synchronization problem. A group of philosophers is sitting around a dining table, and in front of each philosopher having a bowl of spaghetti. Between each pair of adjacent philosophers is a fork. The problem is to construct a solution that avoids deadlock and resource contention.

Here's a representation of a solution:

1. Concurrency and Threading:

- The code represents each philosopher as an individual thread, allowing multiple philosophers to execute concurrently.
- The threading module is utilized for creating and managing threads.

2. Resource Allocation and Synchronization:

- The code employs a locking mechanism (threading.Lock) to ensure mutual exclusion, preventing multiple philosophers from accessing shared resources simultaneously.
- Conditions (threading.Condition) are used to coordinate the synchronization between philosophers, allowing them to communicate and signal each other based on their states.

3. Philosopher States:

- The state of each philosopher is represented by the state list, where 0 indicates thinking, 1 indicates being hungry, and 2 indicates eating.
- Transitions between states are managed to prevent conflicts and ensure that no two adjacent philosophers are eating at the same time.

4. Functions:

- **think(id):** Simulates the philosopher thinking for a random duration.
- **pickup_forks(id):** Attempts to acquire forks for eating while considering the states of adjacent philosophers. Uses conditions for synchronization and waits if necessary.
- **eat(id):** Simulates the philosopher eating for a random duration.

- **return_forks(id):** Releases the forks and signals adjacent philosophers to check if they can start eating.

5. Philosopher Routine:

- The philosopher function encapsulates the main routine for each philosopher, executing a specified number of iterations of thinking, eating, and returning forks.

6. Main Execution:

- The code block under if __name__ == "__main__": creates threads for each philosopher, starts them concurrently, and then waits for all philosopher threads to complete using the join method.

This solution ensures that the philosophers alternate between thinking and eating, with proper synchronization to avoid conflicts and deadlocks. It addresses the dining philosopher's problem by providing a concurrent and synchronized solution to the challenges of resource sharing in a dining hall. Each philosopher completes one iteration of thinking, attempting to eat, and returning forks. The variable **num_iterations** are set to 1, defining a short demonstration of the cyclic nature of philosopher activities. Adjusting this value allows for control over the number of iterations, influencing the overall execution and behavior of the scenario.

However, with this solution philosophers can only eat if they have both forks It also avoids competition for resources by locking mechanisms for forks.

Code Snippet

1) Code Setup (Section 1):

```
import threading
import time
import random

NUMPHIL = 5
num_iterations = 1
LEFT = lambda i: (i - 1 + NUMPHIL) % 5
RIGHT = lambda i: (i + 1) % 5

state = [0] * NUMPHIL
identity = [0, 1, 2, 3, 4]
lock = threading.Lock()
cond = [threading.Condition(lock) for _ in range(NUMPHIL)]
```

Import 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.

Import time:

The **time.sleep(seconds)** function form time module in Python pauses the current thread's execution for a specified duration. It serves as a valuable tool for introducing intentional delays or regulating the timing of actions within a program, enhancing control over the flow of execution. This functionality is particularly useful in scenarios where time-based intervals are needed for synchronization in the program.

Import random

The **random** module in this code introduces variability in the duration of both the thinking and eating phases of the philosophers. By generating random time intervals for thinking and eating using **random.randint(1, 3)**, the code simulates a more dynamic and realistic scenario.

Choice of Indices:

Indices are chosen in such a way that the philosopher to the left of a given philosopher with index **i** is **(i - 1 + NUMPHIL) % NUMPHIL**, and the philosopher to the right is **(i + 1) % NUMPHIL**. This ensures that the indices wrap around in a circular manner, simulating the circular arrangement of philosophers at a dining table.

Lambda Functions:

- The lambda functions are used to create functions (**LEFT** and **RIGHT**) that calculate the indices of philosophers to the left and right of a given philosopher.
- The functions take an argument **i**, which represents the index of the current philosopher, and return the calculated indices

state = [0] * NUMPHIL

It is used to initialize the state of all philosophers to zero. In the context of your dining philosopher's problem, the state list represents the current state of each philosopher. In this case, the value 0 might be associated with the state of "thinking" or an initial/default state.

identity = [0, 1, 2, 3, 4]:

This line creates a list named **identity** that contains the unique indices assigned to each philosopher. In the dining philosopher's problem, each philosopher is identified by a unique index, and these indices are stored in the **identity** list.

lock = threading. Lock():

This line creates a threading lock named **lock**. A lock is used to synchronize access to shared resources in a multithreaded environment. It ensures that only one thread can access the critical section (protected by the lock) at a time, preventing data corruption or conflicts.

cond = [threading.Condition(lock) for _ in range(NUMPHIL)]

In this code, there's a list called cond that holds special signals (condition variables). Imagine these signals like a way for the philosophers to talk to each other. For each philosopher (there are NUMPHIL philosophers), we create one of these signals linked with a lock. The lock helps to make sure that only one thing happens at a time, avoiding confusion. All these signals are put together in the cond list. So, it's like a shared space where the philosophers can signal each other, making sure things happen in an organized way when they're all trying to eat and think together.

2) Code Explanation (Section 2):

```python
def think(id):
    think_time = random.randint(1, 3)
    print(f"Philosopher {id} is thinking for {think_time} seconds")
    time.sleep(think_time)
    print(f"Philosopher {id} reappears from sleep from thinking")

def pickup_forks(id):
    left = LEFT(id)
    right = RIGHT(id)
    with lock:
        state[id] = 1  # HUNGRY

        while state[id] == 1 and (state[left] == 2 or state[right] == 2):
            print(f"Philosopher {id} is hungry and waiting to pick up forks to
eat")
            cond[id].wait()

        if state[id] == 1:
            state[id] = 2  # EATING
            print(f"Philosopher {id} is allowed to eat now")
```

<u>Think Function:</u>

1. **Think Function Definition:**
    - The code defines the **think** function to simulate a philosopher's thinking process.
2. **Function Parameter - Philosopher Identifier:**
    - The function takes one parameter, **id**, representing the unique identifier of the philosopher. This parameter helps identify which philosopher is currently engaged in thinking.
3. **Random Thinking Time Generation:**
    - The line **think_time = random.randint(1, 3)** generates a random integer (between 1 and 3) and assigns it to **think_time**. This variable represents the duration the philosopher will spend thinking.
4. **Print Statements for Thinking Simulation:**
    - **print(f"Philosopher {id} is thinking for {think_time} seconds")** prints a message indicating the start of thinking, including the philosopher's identifier and the thinking time.
    - **time.sleep(think_time)** simulates the thinking time by pausing the program execution.
    - **print(f"Philosopher {id} reappears from sleep from thinking")** prints a message indicating the end of the thinking period, signaling that the philosopher has finished thinking.

<u>Pickup_fork Function:</u>

1. **Neighbor Calculation:**
    - **left = LEFT (id)** and **right = RIGHT (id)** calculates the indices of the philosopher's left and right neighbors. This ensures circular indexing for the philosophers seated around the table.
2. **Lock Acquisition:**
    - The **with lock:** statement is employed to acquire a lock, ensuring that only one philosopher at a time can access critical sections of the code. This prevents race conditions and ensures mutual exclusion during the fork-pickup process.
3. **Hungry State and Waiting Loop:**
    - **state[id] = 1** sets the philosopher's state to 1, indicating hunger. The subsequent **while** loop ensures that the philosopher waits for forks while both its left and right

neighbors are eating (state 2). This prevents conflicts and promotes orderly resource acquisition.

4. **Condition Variable Wait and State Transition:**

   - **cond[id]. wait ()** releases the lock associated with the philosopher's condition variable, allowing other philosophers to proceed while the current philosopher waits. After waking up, the code checks if the philosopher is still hungry (**state[id] == 1**), and if so, the philosopher proceeds to pick up forks, setting its state to eating (state 2). This mechanism ensures that a philosopher only picks up forks when it's allowed to do so, avoiding conflicts and promoting synchronization.

```
def eat(id):
    eating_time = random.randint(1, 3)
    print(f"Philosopher {id} is eating for {eating_time} seconds")
    time.sleep(eating_time)
    print(f"Philosopher {id} reappears from sleep from eating")

def return_forks(id):
    left = LEFT(id)
    right = RIGHT(id)
    with lock:
        state[id] = 0  # THINKING

        print(f"Philosopher {id} has put down forks")
        cond[left].notify()
        print(f"Philosopher {id} signaled philosopher {left} to see if it can
eat")
        cond[right].notify()
        print(f"Philosopher {id} signaled philosopher {right} to see if it can
eat")
```

<u>Eat Function:</u>

1. **Eat Function Definition:**

   - The code defines the **eat** function to simulate the act of a philosopher eating.

2. **Function Parameter - Philosopher Identifier:**

   - The function takes one parameter, **id**, representing the unique identifier of the philosopher. This parameter helps identify which philosopher is currently engaged in eating.

3. **Random Eating Time Generation:**
    - **eating_time = random.randint(1, 3)** generates a random integer (between 1 and 3) and assigns it to **eating_time**. This variable represents the duration the philosopher will spend eating.
4. **Print Statements for Eating Simulation:**
    - **print(f"Philosopher {id} is eating for {eating_time} seconds")** prints a message indicating the start of eating, including the philosopher's identifier and the eating time.
    - **time.sleep(eating_time)** simulates the eating time by pausing the program execution.
    - **print(f"Philosopher {id} reappears from sleep from eating")** prints a message indicating the end of the eating period, signaling that the philosopher has finished eating.

<u>Return Fork Function:</u>

1. **Neighbor Calculation:**
    - **left = LEFT(id)** and **right = RIGHT(id)** calculate the indices of the philosopher's left and right neighbors. This ensures circular indexing for the philosophers seated around the table.
2. **Lock Acquisition:**
    - The **with lock:** statement is employed to acquire a lock, ensuring that only one philosopher at a time can access critical sections of the code. This prevents race conditions and ensures mutual exclusion during the fork-return process.
3. **State Transition to THINKING:**
    - **state[id] = 0** sets the philosopher's state to 0, indicating that the philosopher is now in the thinking state after putting down the forks.
4. **Print Statements and Signaling Neighbors:**
    - **print(f"Philosopher {id} has put down forks")** prints a message indicating that the philosopher with the given id has finished eating and put down its forks.
    - **cond[left].notify()** and **cond[right].notify()** signal the left and right neighbors, respectively, using the notify() method on their associated condition variables. This signals the neighbors to check if they can start eating, facilitating the coordination of actions among philosophers.

3) Code Ending (Section 3):

```python
def philosopher(num, iterations):
    id = num
    for _ in range(iterations):
        think(id)
        pickup_forks(id)
        eat(id)
        return_forks(id)


if __name__ == "__main__":
    philosophers = [threading.Thread(target=philosopher, args=(i, 
num_iterations)) for i in range(NUMPHIL)]

    for phil in philosophers:
        phil.start()

    for phil in philosophers:
        phil.join()
```

1. **Philosopher Function Definition:**

   - The code defines a function named **philosopher** that represents the main routine for each philosopher in the dining philosophers simulation.

2. **Function Parameters - Philosopher Identifier and Iterations:**

   - The **philosopher** function takes two parameters, **num** (philosopher identifier) and **iterations** (number of cycles the philosopher goes through). These parameters guide the behavior of each philosopher during the simulation.

3. **Iterative Routine - Thinking, Eating, and Returning Forks:**

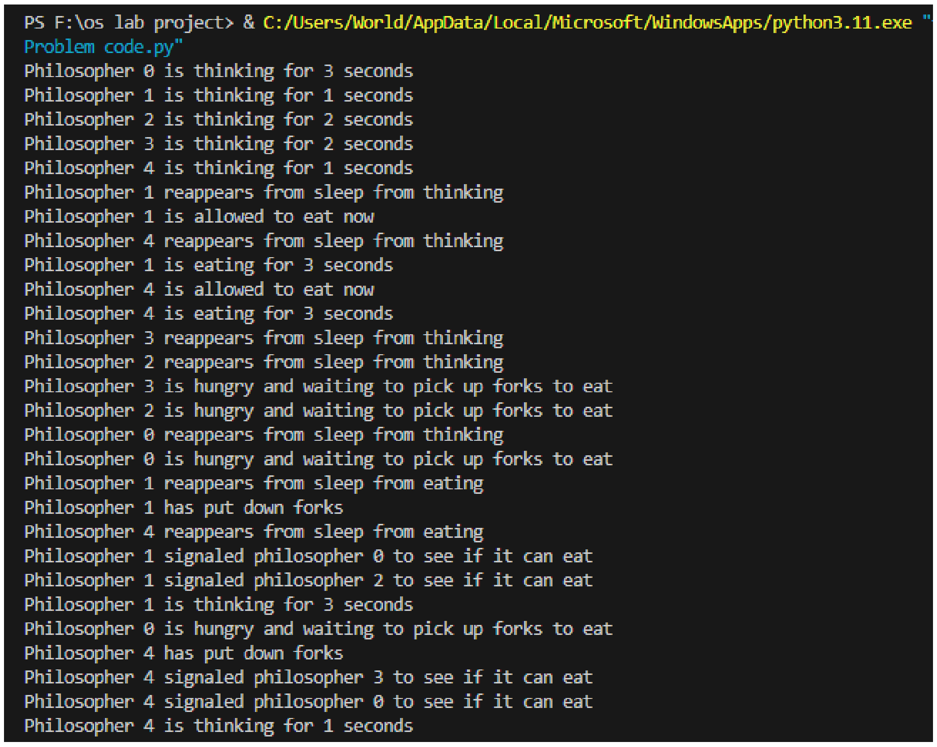
   - Inside the **philosopher** function, there's a loop that iterates **iterations** times. In each iteration, the philosopher engages in thinking, attempts to pick up forks, eats, and then returns the forks. This loop defines the repetitive sequence of actions for each philosopher.

4. **Thread Creation for Each Philosopher:**

   - The **philosopher's** list is created using list comprehension, where each element is a thread representing a philosopher. The **target** parameter is set to the **philosopher** function, and the **args** parameter includes the philosopher's identifier (**i**) and the number of iterations (**num_iterations**).

5. **Concurrent Execution - Thread Start:**
   - A **for** loop is used to start each philosopher thread concurrently using the **start()** method. This allows multiple philosophers to execute their routines simultaneously.
6. **Thread Synchronization - Thread Join:**
   - Another **for** loop with the **phil.join()** statement is used to wait for each philosopher thread to complete its execution. This ensures that the main program doesn't proceed until all philosopher threads have finished their respective routines. The **join()** method helps synchronize the main thread with the philosopher threads.

Outcome:



```
PS F:\os lab project> & C:/Users/World/AppData/Local/Microsoft/WindowsApps/python3.11.exe "
Problem code.py"
Philosopher 0 is thinking for 3 seconds
Philosopher 1 is thinking for 1 seconds
Philosopher 2 is thinking for 2 seconds
Philosopher 3 is thinking for 2 seconds
Philosopher 4 is thinking for 1 seconds
Philosopher 1 reappears from sleep from thinking
Philosopher 1 is allowed to eat now
Philosopher 4 reappears from sleep from thinking
Philosopher 1 is eating for 3 seconds
Philosopher 4 is allowed to eat now
Philosopher 4 is eating for 3 seconds
Philosopher 3 reappears from sleep from thinking
Philosopher 2 reappears from sleep from thinking
Philosopher 3 is hungry and waiting to pick up forks to eat
Philosopher 2 is hungry and waiting to pick up forks to eat
Philosopher 0 reappears from sleep from thinking
Philosopher 0 is hungry and waiting to pick up forks to eat
Philosopher 1 reappears from sleep from eating
Philosopher 1 has put down forks
Philosopher 4 reappears from sleep from eating
Philosopher 1 signaled philosopher 0 to see if it can eat
Philosopher 1 signaled philosopher 2 to see if it can eat
Philosopher 1 is thinking for 3 seconds
Philosopher 0 is hungry and waiting to pick up forks to eat
Philosopher 4 has put down forks
Philosopher 4 signaled philosopher 3 to see if it can eat
Philosopher 4 signaled philosopher 0 to see if it can eat
Philosopher 4 is thinking for 1 seconds
```

The result of the dining philosophers code shows that the philosophers can work together without causing problems. They do things at the same time without getting in each other's way. Adding randomness makes it more realistic, like real people having different paces for eating and thinking.

Results and Conclusion

The implementation of the Dining Philosophers problem using the Python demonstrates the following outcomes:

Results:

1. **Concurrent Execution:**
    - The code demonstrates concurrent execution by creating individual threads for each philosopher. This allows multiple philosophers to perform their actions simultaneously, such as thinking, attempting to eat, and returning forks.
2. **Synchronization and Coordination:**
    - The use of locks and condition variables ensures synchronization and coordination among philosophers. This prevents conflicts and race conditions, especially during critical sections where philosophers pick up or return forks.
3. **Randomized Behavior:**
    - The incorporation of random durations for thinking and eating adds variability to the simulation. Each philosopher spends different amounts of time in these states, making the simulation more dynamic.

Conclusions:

1. **Concurrency Challenges Addressed:**
    - The code successfully addresses challenges related to concurrent execution by employing locks and conditions. This ensures that philosophers can perform their actions without interference from other philosophers.
2. **Prevention of Deadlocks:**
    - Through the use of locks and careful state management, the code mitigates the risk of deadlocks, a common issue in scenarios with shared resources. Philosophers wait for favorable conditions before proceeding, preventing conflicts.
3. **Realistic Simulation:**
    - The inclusion of random durations for thinking and eating adds a level of realism to the simulation. This randomness introduces unpredictability, reflecting the dynamic nature of real-world scenarios.
4. **Thread Synchronization:**

- The code effectively synchronizes the main program with the philosopher threads using the **join()** method. This ensures that the main program waits for all philosophers to complete their iterations before concluding.

In summary, the code provides a concurrent and synchronized solution to the dining philosophers problem, demonstrating effective coordination among philosophers and addressing common challenges associated with concurrent programming. The randomized durations contribute to a more realistic simulation of the dining philosopher's scenario.

Future work and Applications in Operating systems

Resource Sharing Models:

The dining philosopher's problem serves as a fundamental model for resource sharing in operating systems. Understanding and solving such synchronization challenges are crucial for designing efficient resource-sharing algorithms.

Concurrency Control Strategies:

Operating systems leverage concepts from the dining philosopher's problem to implement robust concurrency control strategies. Techniques such as locks, semaphores, and condition variables, inspired by this problem, are foundational in OS design.

Multithreading Concepts:

The dining philosopher's scenario exemplifies the principles of multithreading and concurrent programming. Knowledge gained from addressing synchronization issues in this context is directly applicable to the design and implementation of multithreaded applications in operating systems.

Deadlock Avoidance Techniques:

Solutions to prevent deadlocks in the dining philosopher's problem offer insights into deadlock avoidance techniques in operating systems. Understanding and applying these principles contribute to the development of reliable and deadlock-free systems.

References

[1] Valluri, C. (2022, November 13). The Dining Philosophers Problem Project. GitHub. https://github.com/chanakyav/The-Dining-Philosophers-Problem-Project

[2] Anna, D. (2020, October 24). *The Dining Philosopher's Problem*. Medium. https://medium.com/swlh/the-dining-philosophers-problem-bbdb92e6b788