MANISA CELAL BAYAR UNIVERSITY

# Parallel Programming
# Dining Philosophers Problem and It’s Solution by Token Ring Approach

Name(s): Akif Tunç, Ömer Taylan DURUK

Number(s): 190315078, 190315059

## 1- Dining Philosophers Problem

In computer science, the dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

It was originally formulated in 1965 by Edsger Dijkstra as a student exam exercise, presented in terms of computers competing for access to tape drive peripherals. Soon after, Tony Hoare gave the problem its present form.[1][2][3][4]

The main purpose of the Dining Philosophers problem is to illustrate and address challenges related to concurrent programming, resource allocation, and synchronization.

Key purposes of the Dining Philosophers problem include:

a) Concurrency Challenges: The problem demonstrates the complexities and challenges of managing multiple processes concurrently. In a system where several entities (philosophers) are competing for shared resources (chopsticks), it highlights the need for careful synchronization to avoid conflicts.
b) Deadlock Prevention: One of the primary goals of solving the Dining Philosophers problem is to prevent deadlocks. Deadlocks occur when processes are blocked indefinitely because each is waiting for a resource held by another. The problem encourages the development of strategies to ensure that all philosophers can make progress without getting stuck in a deadlock.
c) Resource Allocation: The philosophers and chopsticks represent concurrent processes and shared resources, respectively. The problem emphasizes the importance of efficient resource allocation, where philosophers can access the resources, they need without causing conflicts or resource contention.
d) Synchronization Techniques: Solving the Dining Philosophers problem often involves the use of synchronization techniques, such as mutexes or semaphores. It serves as an exercise in understanding and implementing these techniques to coordinate the activities of concurrent processes.
e) Algorithmic Solutions: Various algorithms can be devised to solve the Dining Philosophers problem, and exploring these solutions contributes to the development of algorithms for managing shared resources in concurrent systems.
f) Educational Purpose: The problem is frequently used in computer science education to teach students about concurrency issues, parallel programming, and the challenges of designing systems that involve multiple interacting components.
g) Real-world Analogy: The scenario of philosophers sitting around a dining table, alternating between thinking and eating, is a metaphor for real-world scenarios where multiple entities need to share limited resources without causing conflicts or inefficiencies.

By exploring and solving the Dining Philosophers problem, programmers and students gain insights into fundamental concepts of concurrent programming and develop skills that are applicable in a wide range of computing scenarios, including operating systems, distributed systems, and parallel programming environments.

## 2- The Problem Statement

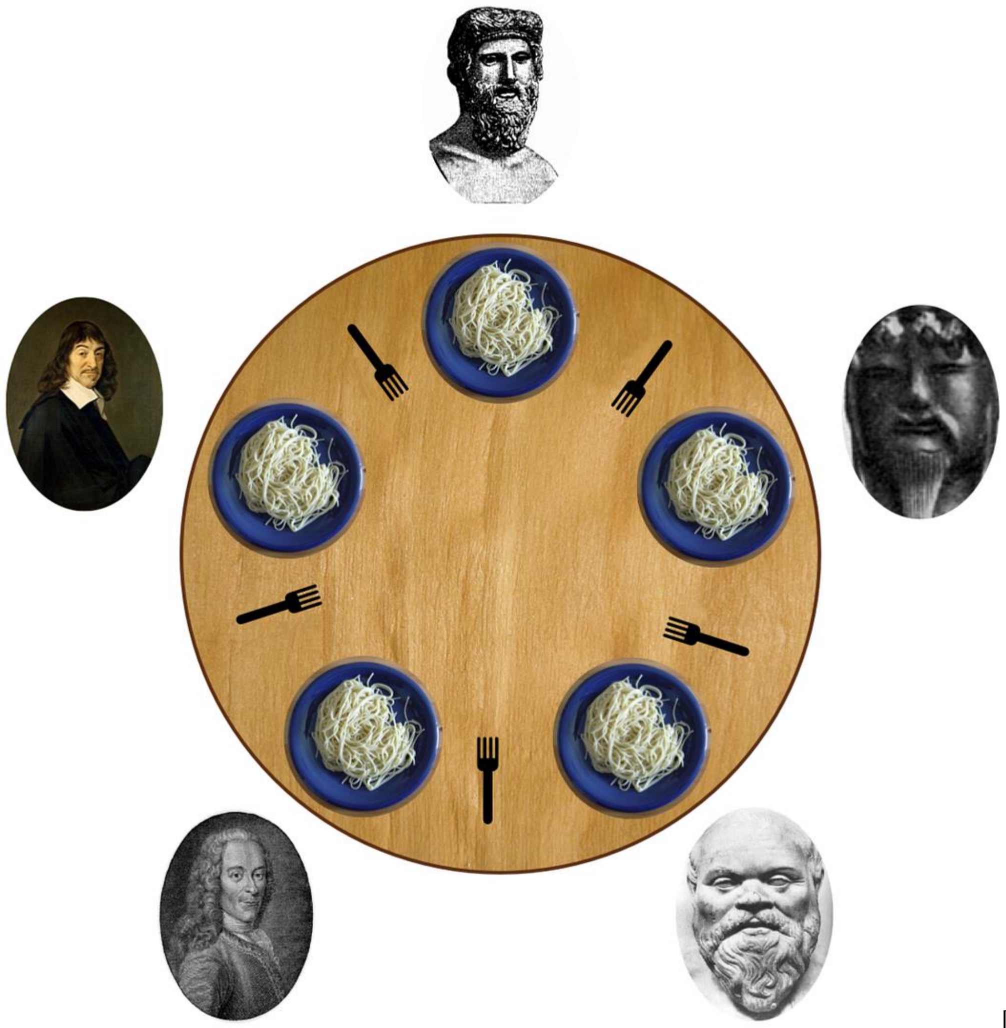
Five philosophers dine together at the same table. Each philosopher has his own plate at the table. There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat his spaghetti when he has both a left and right fork. Thus two forks will only be available when his two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, he will put down both forks. The problem is how to design a regimen (a concurrent algorithm) such that any philosopher will not starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think (an issue of incomplete information).

**Problems**

The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. To see that a proper solution to this problem is not obvious, consider a proposal in which each philosopher is instructed to behave as follows:

- think unless the left fork is available; when it is, pick it up;
- think unless the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- put the left fork down;
- put the right fork down;
- repeat from the beginning.

However, each philosopher will think for an undetermined amount of time and may end up holding a left fork thinking, staring at the right side of the plate, unable to eat because there is no right fork, until he starves.



[5]

## 3- Solution Approaches

1. Mutex (Semaphore) Solution:
    a. Idea: Uses a semaphore (or mutex) for each chopstick. Philosophers acquire both left and right chopsticks before eating to avoid conflicts.

    b. Explanation: Each philosopher tries to acquire the mutex (or semaphore) for the left and right chopsticks. If successful, they eat and then release the acquired chopsticks.

2. Resource Hierarchy Solution:
    a. Idea: Introduces a hierarchy among the chopsticks. Philosophers must acquire chopsticks in a specific order (e.g., always left before right).
    b. Explanation: Philosopher i first acquires a global mutex to access shared variables. Then, they determine the left and right chopsticks based on their position. The philosopher acquires the left and right chopsticks in a specified order, eats, and releases the chopsticks.[6]
3. Chandy/Misra Solution:
    a. Idea: Uses a central authority (arbiter) to manage access to chopsticks. Philosophers request permission from the arbiter before picking up the chopsticks.
    b. Explanation: Philosophers request chopsticks through a central arbiter, indicating their intention to eat. The arbiter grants permission only if the requested chopsticks are available. The philosophers then eat and release the chopsticks.
4. Semaphore Solution:
    a. Idea: Uses semaphores for chopsticks. Philosophers acquire both left and right chopsticks using semaphores.
    b. Explanation: Similar to the Mutex Solution, but it uses semaphores explicitly for synchronization. Philosophers acquire semaphores for both left and right chopsticks, ensuring exclusive access to the shared resource.
5. Asymmetric Solution:
    a. Idea: Introduces asymmetry in resource access. Philosophers first acquire a global mutex and then a separate semaphore for the eating philosopher to control access.
    b. Explanation: Philosopher i first acquires a global mutex to access shared variables. Then, they acquire a separate semaphore to ensure exclusive access to the eating process. This approach introduces an asymmetry in access to resources.
6. Timeout Solution:
    a. Idea: Introduces a timeout mechanism to prevent philosophers from waiting indefinitely. If a philosopher cannot acquire both chopsticks within a timeout, they reconsider their strategy.
    b. Explanation: Philosophers attempt to acquire chopsticks, and if unsuccessful within a specified timeout, they reconsider their strategy. This helps prevent deadlock situations by allowing philosophers to try different approaches.
7. Resource Hierarchy Solution:
    a. Idea: Introduces a resource hierarchy. Philosophers request resources based on a priority determined by a resource hierarchy.
    b. Explanation: Philosophers request resources based on a priority determined by a resource hierarchy. The hierarchy is updated dynamically, and philosophers eat only if they successfully acquire the highest-priority resources. This introduces a mechanism for resource allocation based on priority.
8. Token Ring Solution:
    a. Idea: A token is passed among the philosophers in a ring. A philosopher can eat only when they possess the token. After eating, they pass the token to the next philosopher in the ring.
    b. Explanation: Each philosopher is initially in the thinking state. To eat, a philosopher needs to possess the token. Philosophers follow a protocol to request and release the token, ensuring mutual exclusion. The token circulates in a ring, and only the philosopher possessing the token can enter the critical section (eat). After eating, the philosopher releases the token, allowing the next philosopher in the ring to acquire it and enter the critical section.

## 4- The Token Ring Solution:

The Token Ring solution for the Dining Philosophers problem introduces a token or permission mechanism to regulate access to the critical section (eating). The idea is to pass a token among the philosophers in a ring-like structure, and a philosopher can eat only when they possess the token. After eating, the philosopher passes the token to the next philosopher in the ring. This approach ensures that only one philosopher can eat at a time, preventing conflicts and ensuring mutual exclusion.

Let's go through the details of the Token Ring solution:

**Components:**

1. Mutex (Semaphore): A global mutex is used to control access to shared variables and ensure that the token passing mechanism is synchronized.

```
mutex = Semaphore(1)
```

2. Token Semaphore: A semaphore (or token) is used to manage the possession of the token. The initial value of the semaphore indicates whether a philosopher possesses the token or not.

```
token = Semaphore(0)  # Initially, no philosopher possesses the token
```

**Philosopher Process:**

Each philosopher follows a protocol to request and release the token, ensuring that they can enter the critical section (eat) only when they possess the token.

```
def philosopher(i):
    while True:
        think()
        mutex.acquire()
        wait_for_token(i)  # Wait for the token
        eat()
        pass_token(i)  # Pass the token to the next philosopher
        mutex.release()
```

1. `wait_for_token(i)` Function:

This function is responsible for a philosopher waiting until they possess the token.

```
def wait_for_token(i):
    mutex.release()     # Release the global mutex
    token.acquire()     # Wait until the token is acquired
```

Here, the philosopher releases the global mutex, allowing other philosophers to progress, and then waits until they acquire the token. If a philosopher doesn't have the token, they will be blocked until the token is released.

2. `pass_token(i)` Function:

After eating, the philosopher passes the token to the next philosopher in the ring.

```
def pass_token(i):
    token.release()  # Pass the token to the next philosopher
```

Here, the philosopher releases the token semaphore, indicating that they are done with the critical section (eating). This allows the next philosopher in the ring to acquire the token and proceed.

**Advantages:**

1. Mutual Exclusion: Only one philosopher can eat at a time since the token is required for entry into the critical section.

2. Simplicity: The Token Ring solution is conceptually simple and easy to understand.

**Considerations:**

1. Overhead: The token passing mechanism introduces a delay, and philosophers must wait for their turn. This may lead to inefficiency if philosophers are not frequently requesting to eat.

2. Unnecessary Token Passing: If a philosopher is not hungry, they may unnecessarily pass the token, potentially introducing additional overhead.

The Token Ring solution provides a straightforward approach to solving the Dining Philosophers problem, emphasizing simplicity and mutual exclusion. However, its efficiency may be influenced by the specific characteristics of the system and the frequency at which philosophers request access to the critical section.

## 5- References:

1. Dijkstra, Edsger W. EWD-1000 (PDF). E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (transcription)

2. ^ Jump up to:[a] [b] J. Díaz; I. Ramos (1981). Formalization of Programming Concepts: International Colloquium, Peniscola, Spain, April 19–25, 1981. Proceedings. Birkhäuser. pp. 323 , 326. ISBN 978-3-540-10699-9.

3. ^ Hoare, C. A. R. (2004) [originally published in 1985 by Prentice Hall International]. "Communicating Sequential Processes" (PDF). usingcsp.com.

4. ^ Tanenbaum, Andrew S. (2006), Operating Systems - Design and Implementation, 3rd edition [Chapter: 2.3.1 The Dining Philosophers Problem], Pearson Education, Inc.

5. Benjamin D. Esham / Wikimedia Commons CC BY-SA 3.0, via Wikimedia Commons.

6. Tanenbaum, Andrew S. (2006), Operating Systems - Design and Implementation, 3rd edition [Chapter: 3.3.5 Deadlock Prevention], Pearson Education, Inc.