Dining philosophers problem in operating system

Chu Trung Anh
anh.ct225564@sis.hust.edu.vn

Tran Nam Tuan Vuong
vuong.tnt225540@sis.hust.edu.vn

Vu Minh Hieu
hieu.vm225494@sis.hust.edu.vn

***Abstract*—The Dining Philosophers Problem is a classic synchronization issue in computer science. It models the complexities of resource allocation and process synchronization in a concurrent computing environment. This paper explores the theoretical basis of the problem, its significance in understanding deadlock, livelock and starvation to achieve efficient synchronization. By analyzing different approaches such as resource hierarchy, semaphore, limit number od dinners, Chandy/Misra. The report aims to provide a comprehensive overview of the problem and its real-world implications in system design.**

To see the detail code implementation, visit our GitHub repository at this link

***Index Terms*—Operating system, Dining philosophers, Synchronization problems, Multi-thread**

I. Introduction

Concurrency and synchronization are fundamental aspects of computer science, particularly in operating systems and distributed computing. The Dining Philosophers Problem, introduced by Edsger Dijkstra in 1965, serves as a classic example of the challenges involved in managing shared resources in concurrent systems while avoiding issues like deadlock and starvation.

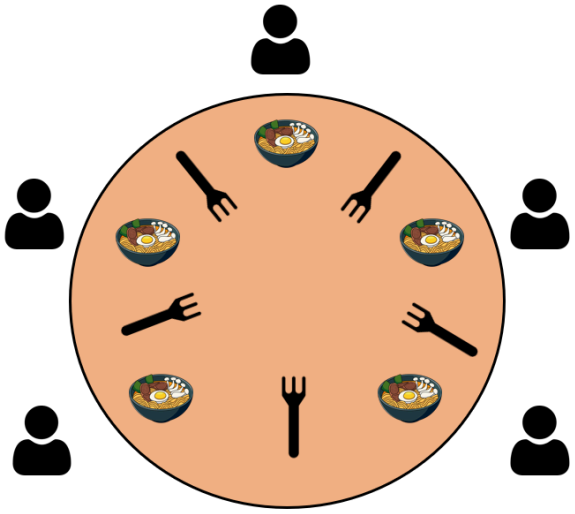
This problem, though simple in structure, encapsulates key concepts such as resource contention, deadlock, and fairness. It is widely used as a pedagogical tool for understanding synchronization techniques and evaluating strategies for deadlock prevention. While the problem itself is theoretical, its principles have practical relevance, offering insights into challenges encountered in areas such as database systems, distributed computing, and networking, where shared resource management and process synchronization are critical for maintaining efficient, reliable systems. . This report focuses on the theoretical underpinnings of the problem and examines various strategies to address its challenges.

A. *Problem Describe*

The Dining Philosophers Problem illustrates a scenario where five philosophers sit around a circular table. Each philosopher has their own place 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 philosophe continuous alternating between two activities: **thinking** and **eating**

- Thinking state: In this state, the philosopher simply sits and does nothing.
- Eating state: To eat, a philosopher needs to pick up two forks, one at a time. He takes the fork on the left hand side first, then the right ones. Only when he can obtain both forks, he can start to eat. The process is as follows:
  1) The philosopher first tries to pick up the fork on their left-hand side. If it is unavailable (because another philosopher is using it), they must wait until it becomes free.
  2) Once they successfully pick up the left fork, they then attempt to pick up the fork on their right-hand side. If the right fork is also unavailable, they must wait until it becomes free. **During this waiting period, the philosopher does not release the left fork he is holding for others to use**
  3) Once obtaining both forks, the philosopher can begin eating
  4) When finished eating, the philosopher releases the forks in reverse order: first the right fork, then the left fork.

Each philosopher alternates between the thinking and eating states, with the duration of each state being unpredictable and occurring randomly. The primary challenge is to design a concurrent algorithm that ensures no philosopher will starve, even under varying conditions.



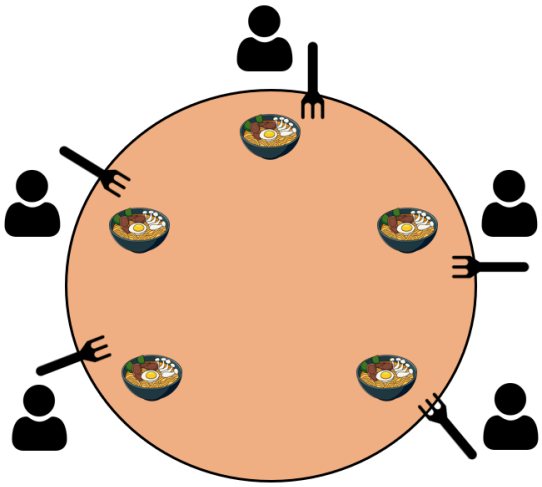
In summary, the Dining Philosophers Problem operates under certain constraints. However, some of the solutions we propose below may slightly relax one or more of these constraints to improve synchronization flexibility and efficiency.

From an operating system's perspective, this problem models resource allocation and process synchronization. The five forks represent five critical resources, each of which can only be used by one philosopher at a time (i.e., sharing capability equals one). The five philosophers represent processes or threads, and their critical section corresponds to the eating state, during which they require exclusive access to two resources (the forks).

B. *Challenge of the problem*

The primary challenge is ensuring all philosophers can eat without encountering issues such as:

- **Deadlock**: occurs when all philosophers are waiting indefinitely for resources that are being held by others, creating a circular dependency. For instance, if all philosophers pick up the fork on their left simultaneously, then all need to wait for the fork on their right, no philosopher will be able to proceed to the eating state. This situation leads to a system freeze where no progress is possible.



- **Livelock**: Livelock is a situation where philosophers keep acting (e.g., picking up and putting down forks) but make no actual progress toward eating. Unlike deadlock, the system is not entirely frozen; however, it is stuck in a repetitive cycle of futile actions. Imagine, all five philosophers picking up their left forks at once, then seeing they can't get their right forks because their neighbors are holding them. They all put down their forks at the same time, wait for a moment, and then try again at exactly the same time. This pattern keeps repeating forever - everyone is busy moving their forks up and down, but nobody ever gets to eat.
- **Starvation**: Starvation occurs when one or more philosophers are unable to eat because others monopolize the forks. While the system may not be deadlocked or livelocked, certain philosophers may wait nearly indefinitely if the resource allocation strategy is unfair. For instance, a philosopher might repeatedly miss opportunities to pick up forks because adjacent philosophers continuously take precedence.

In this report, we primarily focus on addressing the deadlock situation in the Dining Philosophers Problem. Deadlock in a system can occur only when all four of the following conditions are met simultaneously:

1) **Mutual exclusion**: Each resource is non-shareable, meaning only one process can use a resource at a time.
2) **Hold and wait**: A process is holding at least one resource while requesting additional resources that are currently being held by other processes.
3) **No preemption**: Resources cannot be forcibly taken away from a process; they must be released voluntarily by the process holding them.
4) **Circular wait**: There exists a cycle of processes where each process is waiting for a resource that is being held by the next process in the cycle, creating a deadlock loop.

To prevent deadlock, system designers typically address one or more of these conditions to break the cycle and ensure smooth resource allocation.

II. Methods

A. *Resource Hierarchy Solution*

This solution to the dining philosophers problem was originally proposed by Dijkstra. It assigns a partial order to the resources (the forks, in this case) and establishes a convention: all resources must be requested in order, and no two resources unrelated by this order can ever be used by a single philosopher simultaneously.

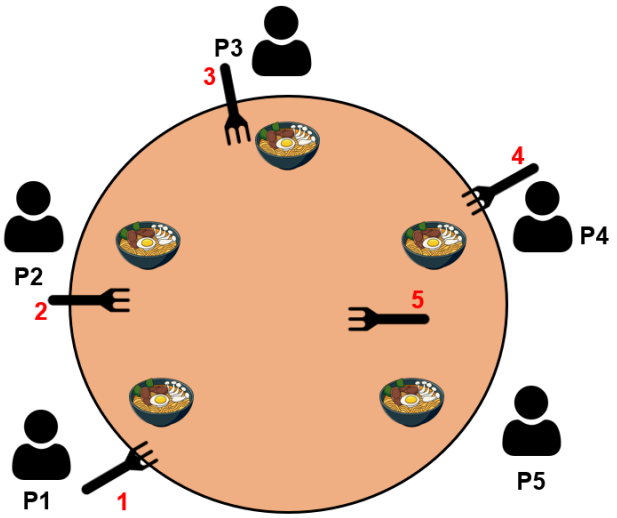
In this approach, the forks are numbered from 1 to 5. Each philosopher always picks up the lower-numbered fork first, followed by the higher-numbered fork from the two forks they need. For example:

- Philosopher P1, seated between forks F1 and F2, picks up fork F1 first, then fork F2.
- Philosopher P3, seated between forks F3 and F4, picks up fork F3 first, then fork F4.

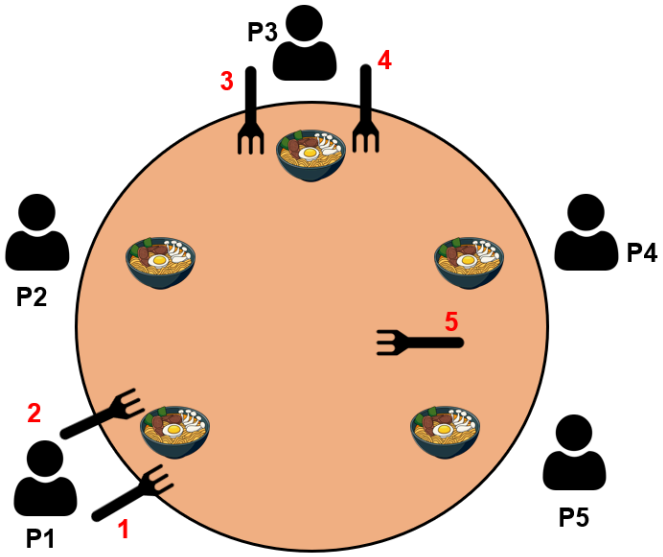
The key idea of this solution is to break the fourth condition of deadlock, circular waiting, by enforcing the resource hierarchy. Let's consider two situations:

- When a philosopher cannot acquire both forks:
  For instance, suppose philosopher P1 initiates the request and picks up fork F1 (since F1 < F2). Meanwhile, happen simultaneously:
  - Philosopher P2 picks up fork F2 (since F2 < F3).
  - Philosopher P3 picks up fork F3 (since F3 < F4).
  - Philosopher P4 picks up fork F4 (since F4 < F5).
  - Philosopher P5 cannot pick up fork F5 because F5 > F1, so he must pick the F1 first. However, fork F1 is currently in use.

Consequently, fork F5 remains free, allowing philosopher P4 to pick it up. This scenario prevents a circular deadlock from forming.



- When a philosopher can acquire both forks: For example:
  - Philosopher P1 picks up fork F1, then fork F2.
  - Philosopher P2 cannot pick up any forks, as they need fork F2 first, which is already in use.
  - Philosopher P3 picks up fork F3, then fork F4.
  - Philosophers P4 and P5 cannot pick up any forks, as the forks they need are already in use.



This solution also ensure fairness: Philosophers get access to the forks in a fair manner, as they all follow the same rule. However, a philosopher might still experience starvation if other philosophers are continuously able to eat. This can be addressed using additional techniques, such as timeouts or a more complex algorithm (e.g., using semaphores or monitors).

Here's a pseudo-code representation of how the Resource Hierarchy Solution can be implemented for the Dining Philosophers Problem.

```
FUNC PHILOSOPHER(i):
1  while True():
2      think()
3      pick_up_fork(min(i, (i + 1) % 5))
4      pick_up_fork(max(i, (i + 1) % N))
5      eat()
6      put_down_fork(max(i, (i + 1) % N))
7      put_down_fork(min(i, (i + 1) % N))
```

While the resource hierarchy effectively prevents deadlocks, it has significant drawbacks that make it impractical in many real-world scenarios:

- **Impracticality in Dynamic Scenarios:** This method assumes that the list of required resources is known in advance, which is often unrealistic. For instance, if a process currently holds resources 3 and 5 (unlike the philosophical example of picking up two forks on either side, a process in practical can hold any set of resources e.g., resources 3 and 5) but decides it needs resource 2, it must: Release resource 5 -> Release resource 3 (the release order in doesn't matter) -> Acquire resource 2 -> Reacquire resources 3 and 5 again. Such operations are time-consuming and inefficient, particularly in systems where releasing and reacquiring resources disrupts workflows or impacts database records.
- **Unfairness:** The solution does not guarantee fairness. For example, if one philosopher (or process) is slow to pick up a fork, while another is quick to think and reclaim forks, the slower philosopher may starve indefinitely. A fair solution must ensure that all processes (or philosophers) eventually access the required resources, regardless of their speed.

These issues make the resource hierarchy solution impractical and inefficient in many real-world applications.

B. Semaphore

Another solution to this problem is using semaphore. Each fork is treated as a critical resource, represented by a semaphore fork[i]. We initialize an array of semaphores as fork[5] = {1, 1, 1, 1, 1} to indicate each fork can only be used by one philosopher at a time.

The pseudo code is as follow:

```
FUNC PHILOSOPHER(i):
1  while True():
2      wait(fork[i])                    // wait left fork
3          wait(fork[(i+1)%5])          // wait right fork
4              eat()
5          signal(fork[(i+1)%5])
6      signal(fork[i])
7      think()
```

This ensures mutual exclusion, where only one philosopher can use a fork at any given time.

However, in case all five philosophers take their left fork simultaneously, a deadlock occurs as no one can proceed to pick up the right fork (as described in the deadlock section).

So we can improve to this solution by introducing an additional semaphore mutex, which ensures that only one philosopher can attempt to pick up forks at a time.

The main idea is:

```
FUNC PHILOSOPHER(i):
1   while True():
2       wait(mutex)
3           wait(fork[i])
4           wait(fork[(i+1)%5])
5       signal(mutex)
6           eat()
7       signal(fork[(i+1)%5])
8       signal(fork[i])
9       think()
```

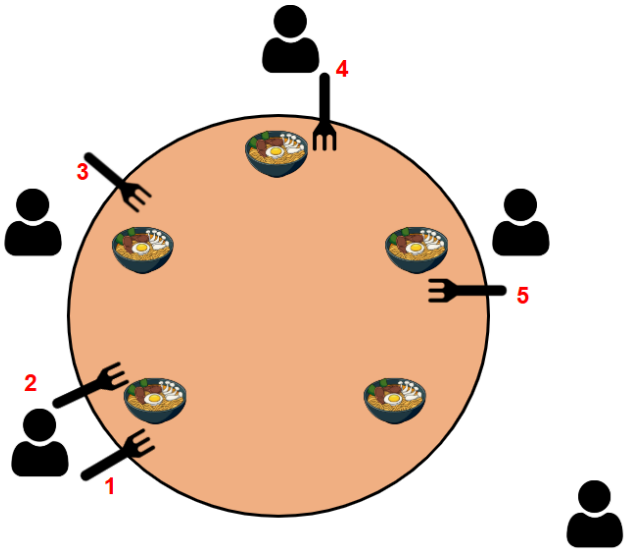
How can this improvement prevent the deadlock?

For example, if all philosophers call wait(mutex) simultaneously, only one gets access. This philosopher can proceed to pick up both forks, eat, and then release the mutex, allowing the next philosopher to act.

However, there are still situations where two non-adjacent philosophers cannot eat simultaneously. For instance, if Philosopher 1 is eating (holding forks 1 and 2), Philosopher 2 may acquire the mutex but fail to get fork 2. Meanwhile, Philosopher 2 still blocks access to the mutex, preventing Philosopher 3 (who has fork 3 and 4 are still free) from proceeding. This violates the progress condition, as philosophers with available resources may remain unable to act.

C. *Limiting the Number of Diners*

One simple solution to avoid deadlock is to limit the number of philosophers allowed to sit at the table at any given time to $n-1$ where n is the total number of philosophers. This solutions is proposed by William Stallings. By implementing this restriction, at least one philosopher will always be able to acquire both the left and right forks to begin eating, effectively preventing a deadlock scenario.



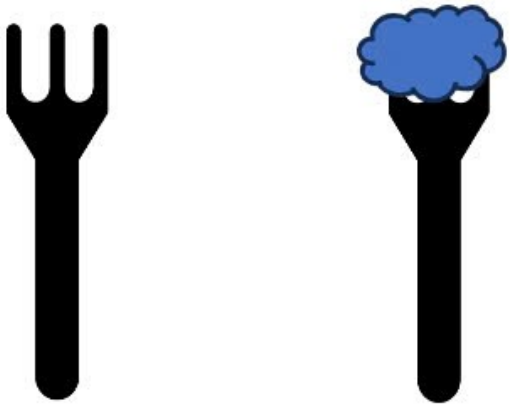
To achieve this, we implement a CountingSemaphore to limit the numbers of philosopher concurrently eating to $n-1$ and n BinarySemaphore for controlling the access of critical resources, in this cases the forks.

```
1   Mutex* forks[n]
2   CountingSemaphore wait_to_sit = n
3   for True:
4       Philosopher [i] is thinking
5       acquire wait_to_sit
6       acquire left fork
7       acquire right fork
8       Philosopher [id] is eating
9       release left fork
10      release right fork
11      release wait_to_sit
12      Philosopher [i] has finished eating and left
```

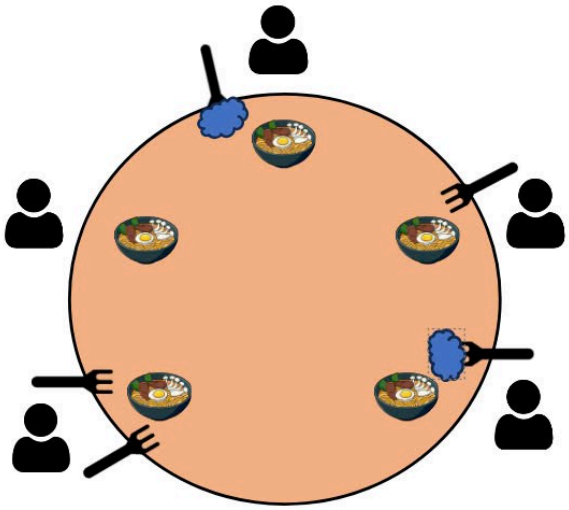
The philosopher who just finished eating will left the table and start thinking, he can requests to return to the table afterwards via the count semaphore.

D. *Chandy/Misra solution*

Another solution to solve the Dinning Philosophers Problem was proposed by K. Chandy and J. Misra in 1984. Chandy-Misra's algorithm can be explained using the concepts of "clean" and "dirty" forks, where each fork is shared between two philosophers.



Each fork is always held by one of its two philosophers. If a fork is dirty, it must be cleaned before being passed to the other philosopher.



Below is the pseudocode of the algorithm:

```
func Philosopher_Routine(philosopher):
1   while True:
2       philosopher.think()        // defer_requests = false
3       philosopher.state =        // defer_requests = true
        "hungry"                   if fork is clean
4       while !
        (philosopher.own_both_forks()):
5           philosopher.request_fork(left_fork)
6           philosopher.request_fork(right_fork)
7       philosopher.eat()          // defer_requests = true
8       philosopher.left_fork.clean
        = false
9       philosopher.right_fork.clean
        = false
10      philosopher.handle_deferred_requests()
```

Initialization: Each philosopher is assigned a unique integer ID. For every pair of philosophers sharing a fork, the fork is assigned to the philosopher with the lower ID (lower neighbor) and marked as "dirty."

- Thinking State: When a philosopher is thinking and receives a request for a fork from a neighbor, the philosopher gives the requested fork to the neighbor, cleaning it beforehand.
- Hungry State: When a philosopher wants to eat, he requests any missing forks from his neighbors. If a neighbor requests a fork during this time: If the fork is dirty, it is cleaned and sent to the neighbor. If the fork is clean, the philosopher keeps it and defers the neighbor's request for later.
- Eating State: A philosopher starts eating only after obtaining all required forks. While eating, all incoming fork requests are deferred, and all forks in possession are marked as dirty.
- Cleanup State: After eating, the philosopher processes any deferred requests for forks, cleaning them before sending them to the requesting neighbors. Once this is done, the philosopher resumes thinking.

Deadlock Avoidance: The Chandy-Misra algorithm avoids deadlock through a directed graph representation. Each philosopher is a vertex, and each fork is an edge with an arrow pointing from "dirty" to "clean." The ID system for philosophers ensures no closed cycles (deadlocks) occur.

No starvation: A hungry philosopher, p, retains all their clean forks. Neighbors must provide their shared forks, cleaned, either immediately (if the neighbor is thinking) or once they finish eating. This ensures p cannot be indefinitely passed over by any neighbor. Over time, all hungry or eating neighbors of p will finish eating, guaranteeing that p will eventually get to eat, though the wait may be long.

Disadvantages: Hungry philosophers may face potentially long wait times due to extended chains of deferred requests.

III. Conclusion

In conclusion, this project thoroughly examined the dining philosophers problem, a fundamental challenge in process synchronization. We explored four distinct solutions: the Resource Hierarchy solution, using Semaphore, the Limiting Number of Diners solution, and the Chandy/Misra solution. Each approach demonstrated unique strengths and weaknesses, showcasing diverse strategies for achieving synchronization and resource allocation in concurrent systems.

To bridge theory and practice, these algorithms were implemented using C++, providing concrete examples of their functionality. Through testing and analysis, we confirmed that all four solutions successfully avoided both deadlocks, thereby empirically validating their correctness and reliability.

This study highlights the importance of selecting appropriate synchronization mechanisms based on system requirements and constraints. By understanding these approaches, developers can design robust and efficient systems that handle resource sharing effectively, contributing to better system performance and stability in real-world applications.

IV. Acknowledgments

We would like to express our heartfelt gratitude to Mr. Do Quoc Huy, our lecturer, for his dedicated teaching of the Operating System course. His detailed guidance and support played a crucial role in helping us complete our project on the Dining Philosophers Problem. This project has been an invaluable learning experience, and we truly appreciate his contributions to our understanding of the subject.

We also wish to thank our friends and families for their unwavering support and encouragement throughout this project. Their understanding and motivation have been essential in helping us manage our time and complete this work successfully.

Team Assignment

- **Chu Trung Anh**: Problem description, challenges, resource hierarchy, semaphore.
- **Vu Minh Hieu**: Challenges, limiting the number of diners.
- **Tran Nam Tuan Vuong**: Chandy/Misra.

V. References

[1] [The Dining Philosophers Problem With Ron Swanson](https://www.adit.io/posts/2013-05-11-The-Dining-Philosophers-Problem-With-Ron-Swanson.html)

[2] [Dining Philosophers Problem - Wikipedia](https://en.wikipedia.org/wiki/Dining_philosophers_problem)

[3] [Dining Philosophers Problem - Belski Blog](https://belski.me/blog/dining-philosophers-problem/)

[4] [Dining Philosophers Problem - YouTube](https://www.youtube.com/watch?v=jCVw7r1INR0&list=PL54DF7EQeBp605B_-ECoKaKo2yEA5Fby5&index=7)

[5] [Dining Philosophers Lecture - University of Colorado](https://home.cs.colorado.edu/~rhan/CSCI_3753_Spring_2005/CSCI_3753_Spring_2005/Lectures/02_22_05_dp_mon_cv.pdf)

[6] [Dining Philosopher Solution Using Monitors] (https://www.naukri.com/code360/library/dining-philosopher-solution-using-monitors)

[7] [Dining Philosophers Solutions - St. Olaf College] (https://www.stolaf.edu/people/rab/pdc/text/dpsolns.htm)