**Dining philosopher’s problem Main idea:**

There are five philosophers that sharing a circular table and they spend their lives by alternating thinking and eating.

Don’t interact with their neighbors.

There is a bowl of ramen for each of the philosophers and five chopsticks.

A hungry philosopher may only eat if there are both chopsticks available.

Otherwise, the philosopher will be on a thinking mode.

**Contents:**

* **Pseudocode**
* **Example of Deadlock**
* **How to solve Deadlock**
* **Example of Starvation**
* **How to solve Starvation**
* **Real world Explanation**

**1. Pseudocode**

1. initialize the states (thinking,eating,hungry)
2. initialize five philosopher
3. put philosophers to nap in “thinking” state

4-for (int i = 0; i < NUM\_OF\_PHILS; i++)

And set state “thinking” for all philosopher

5-if the philosopher need to eat called function Takefork();

6-after the philosopher need to eat and called function

Takefork();,then set lock is locked

7-then set the philosopher state is “hungry”

8-then test if the right and left forks is available and the philosopher is hungry

9-set the philosopher is “eating” , else puts philosopher to

SleepUtilities.nap();

10- when finished eating , called a function Returnforks();

11-and set the state of philosopher “thinking”

12-then letting other philosophers know that forks are available for usage

13-then set lock is unlocked

 **Deadlock:**

**Definition:** A deadlock occurs when two or more threads wait forever for a lock or resource held by another of the threads.

**Deadlock occurs if these conditions are satisfied:**

* Mutual exclusion -- at least one resource must be held by a process.
* Hold and wait -- at least one process holds a resource while it is waiting for another resource.
* No preemption -- one process can't take another process's resources in order to make progress (nor can the OS)
* Circular wait -- there exists a circular chain of processes, each of which is waiting for a resource held by the next process in the chain .

1. **Example of Deadlock:** 
   * + Suppose that all five philosophers become hungry at the same time and each take her left chopstick. All the elements of the chopstick will now be equal to 0.
     + When each philosopher tries to take her right chopstick, she will be delayed forever.
2. **How to solve Deadlock:** 
   * Several possible solutions to the deadlock problem, one of them is to Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).

This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

* + This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher.
  + For this purpose, we introduce the following data structure:

enum

{THINKING, HUNGRY, EATING} state[5];

* + Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating: (state[(i+4) % 5] != EATING) and (state[(i+1) %5] != EATING).
  + We also need to declare condition self\*5+; //wait() and notifyall()
  + This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

 **Starvation:**

**Definition:** Starvation occurs when one or more threads in your program are blocked from gaining access to a resource and, as a result, cannot make progress.

1. **Example of Starvation:**

 The problem is how to design a discipline of behavior

(a concurrent algorithm) such that no philosopher will 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.

 Starvation happens when both philosopher 1 and 3 pick up both chopsticks at the same time and start eating after some time they drop their chopsticks and both philosophers 2 and 5 get to eat . after eating they drop their chopsticks and both philosophers 1 and 3 get to eat again .

Both philosophers 1 and 3 and both philosophers 2 and 5 are taking turn to eat repeatedly .

Hence, philosopher 4 starves to death.

1. **How to solve Starvation:**

The solution for this problem is simply by putting priority on the ones that have waited the longest.

By giving priority, we can prevent starvation from happening

**(6) In dinning philosophe’s Reallife problem we have an example of a bank problem:**

In our example of the dinning philosopher’s problem it will appear between the bank and customers

**As we have here in the bank example:**

1. The customer is: The philosopher, And the bank workers is: The forks.
2. So we arrange the customers by taking a thread number and they will wait.
3. The number of customers equal to the number of workers in the bank.
4. If each customer wants to make a transaction he need 2 workers to get it done.
5. So for example if we have 10 customers and 10 workers, The 10 workers will complete transaction process of 5 customer only and the five other customers each one of them will have to wait until there are 2 workers available for their transaction.
6. The state that the customer will wait we will call it “sleep” state.
7. And if a customer is making a transaction it will be at state “working” and if he finished his state will be “finished”, to leave the opportunity of others customers to make their transaction process.
8. And then when the customer finished the transaction process he will enter into the “thinking” state.
9. And the customer who at the “sleeping” state will enter after that.