**The Dining Philosophers Problem**

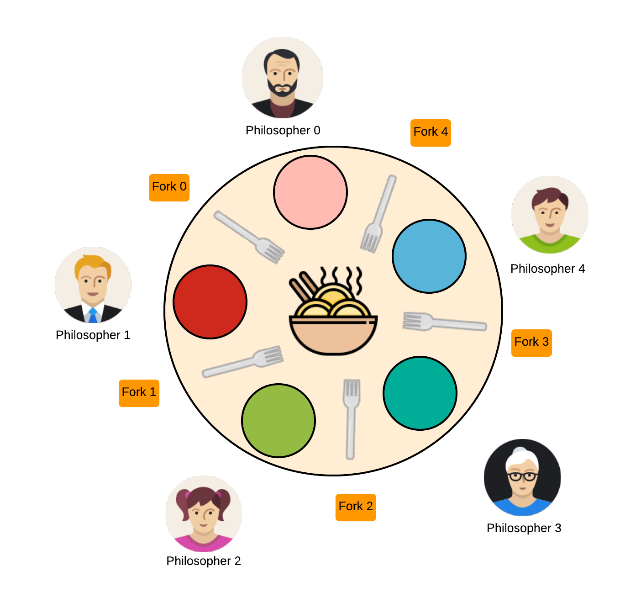
**Aim : To solve the Dining Philosophers problem using Mutex locks or semaphores**

**# Referred from Internet resources**

The Dining Philosophers problem is a classic OS problem that’s usually stated in very non-OS terms:

There are N philosophers sitting around a circular table eating spaghetti and discussing philosophy. The problem is that each philosopher needs 2 forks to eat, and there are only N forks, one between each 2 philosophers. Design an algorithm that the philosophers can follow that ensures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.

The dining philosopher’s problem is invented by E. W. Dijkstra.



ALGORITHM:

Step 1: Start the program.

Step 2: Define the number of philosophers.

Step 3: Declare one thread per philosopher.

Step 4: Declare one chopsticks per philosopher.

Step 5: When a philosopher is hungry.

i. See if chopsticks on both sides are free.

ii. Acquire both chopsticks or.

iii. Eat.

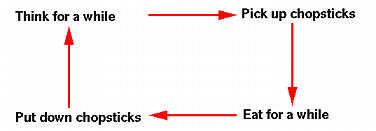
iv. restore the chopsticks.

v. If chopsticks aren’t free.

Step 6: Wait till they are available.

Step 7: Stop the program

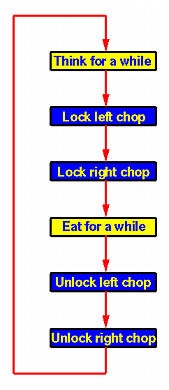
**Race condition**



The "pick up chopsticks" part is the key point. How does a philosopher pick up chopsticks? Well, in a program, we simply print out messages such as ``Have left chopsticks'', which is very easy to do. The problem is each chopstick is shared by two philosophers and hence a ***shared*** resource. We certainly do not want a philosopher to pick up a chopstick that has already been picked up by his neighbor. This is a ***race condition***.

Solution for Race condition

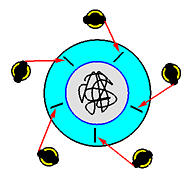
To address this problem, we may consider each chopstick as a shared item protected by a mutex lock. Each philosopher, before he can eat, locks his left chopstick and locks his right chopstick. If the acquisitions of both locks are successful, this philosopher now owns two locks (hence two chopsticks), and can eat. After finishes easting, this philosopher releases both chopsticks, and thinks! This execution flow is shown below.



Because we need to lock and unlock a chopstick, each chopstick is associated with a mutex lock. Since we have five philosophers who think and eat simultaneously, we need to create five threads, one for each philosopher. Since each philosopher must have access to the two mutex locks that are associated with its left and right chopsticks, these mutex locks are global

***Circular Waiting - The most serious problem of this program is that deadlock could occur!***

What if every philosopher sits down about the same time and picks up his left chopstick as shown in the following figure? In this case, all chopsticks are locked and none of the philosophers can successfully lock his right chopstick. As a result, we have a circular waiting (*i.e.*, every philosopher waits for his right chopstick that is currently being locked by his right neighbor), and hence a deadlock occurs.



***Starvation is also a problem!***

Imagine that two philosophers are fast thinkers and fast eaters. They think fast and get hungry fast. Then, they sit down in opposite chairs as shown below. Because they are so fast, it is possible that they can lock their chopsticks and eat. After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat. In this case, the other three philosophers, even though they have been sitting for a long time, they have no chance to eat. This is a ***starvation***. Note that it is not a deadlock because there is no circular waiting, and every one has a chance to eat!

