# OS Assignment 3 - Report for Q1

**By: Arnav Goel (2021519), Section B, Branch: CSAI**

In this question, we are given the dining philosopher's problem, a classic process synchronisation problem, and in the second part, we have been asked to solve a modified version with two soup bowls.

## Q1a - Strict Ordering of Resource Requests:

In this part of the first part of Q1, we were asked to solve the deadlock in the dining philosopher's problem using the strict ordering of resource requests. **I created 5 threads to simulate my 5 philosophers** and passed in the ID of the philosopher as an argument to the thread. I created my lock for this part and **did not use any semaphores or mutex_locks.** My logic for resolving this is that at any point in time, **we allow any ONE philosopher to ask for the forks while the others stay busy waiting.** I have implemented this by using a while loop without a body. **(Something like a spin lock)**. It has conditions to simulate busy waiting such that if 0 is eating, others are waiting, and it ensures strict ordering. For example, **philosopher 1 will start eating only when philosopher 0 has eaten and if and only if, in the last 5 iterations, it didn't eat.**

To simulate this, I use global variables that track whose turn it is. This logic ensures that my philosophers follow strict alternation and eat in the order:
**Phil0 → Phil1 → Phil2 → Phil3 → Phil4 → Phil0 …… and so on.**

## Q1a - Use of Semaphores to access Resources:

This logic uses Djikastra's solution to resolve the deadlock in the given problem. Firstly, I created 5 threads to simulate my 5 philosophers and passed in the ID of the philosopher as an argument to the thread. Then I created 5 binary semaphores to simulate the 5 forks on the table. Now for philosophers 0 to 3, their left fork was their ID, and their right fork was their (ID + 1) % 5. I reverse this definition for philosopher 4, which is what resolves the deadlock. This is because if Philosopher 0 to 3 pick up forks 0 to 3 when Philosopher 4 comes, we would have had a deadlock condition if he had picked up fork 4 too. But according to our change in definition, he would first try to pick up 3, which it would see as occupied and thus would go back in waiting.
This would prevent a deadlock henceforth. We use the sem_wait(), and sem_post() calls to simulate these locks and put the philosophers in waiting.

**Q1b - Semaphores using 2 Bowls:**

This modification would be very easily implemented if we take a bowl semaphore as a counting semaphore with value 2, allowing 2 philosophers to have bowls. We implement this with Dijkstra's algorithm to prevent any deadlock situation by ensuring that the bowls are selected after the forks are.

**Q1b - Strict Ordering with 2 Bowls:**

To implement strict ordering with the two bowls, we simply extend the logic of Question 1a-strict order to this part. We do not need any synchronisation primitives for implementing this as the lock ensures that only one philosopher at any point in time is at the table asking for something to be eaten. Hence, there is no deadlock created even when we need the two bowls on the table for the philosophers to eat.

**Note: I have created 5 threads to simulate the 5 philosophers in all three parts. We have no mutex or semaphores in part a - strict ordering and part b - strict ordering. We use an array of 5 binary semaphores in part a-semaphores to simulate our forks. In part b - in addition to 5 binary semaphores for the forks, I am using a counting semaphore initialised to value 2 for the soup bowl.**
**In all three codes, there is no deadlock on running, and if run for 1000 iterations, all philosophers eat an equal amount of time, indicating that our logic is correct.**