# Laboratory 6

Title of the Laboratory Exercise: Solution to Dining Philosopher problem using Semaphore

1. Introduction and Purpose of Experiment

In multitasking systems, simultaneous use of critical section by multiple processes leads to data inconsistency and several other concurrency issues. By solving this problem students will be able to use semaphore for synchronisation purpose in concurrent programs.

1. Aim and Objectives

Aim

* To develop concurrent programs using semaphores

Objectives

At the end of this lab, the student will be able to

* Use semaphore
* Apply appropriate semaphores in different contexts
* Develop concurrent programs using semaphores

1. Experimental Procedure
   * 1. Analyse the problem statement
     2. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
     3. Implement the algorithm in C language
     4. Compile the C program
     5. Test the implemented program
     6. Document the Results
     7. Analyse and discuss the outcomes of your experiment
2. Question

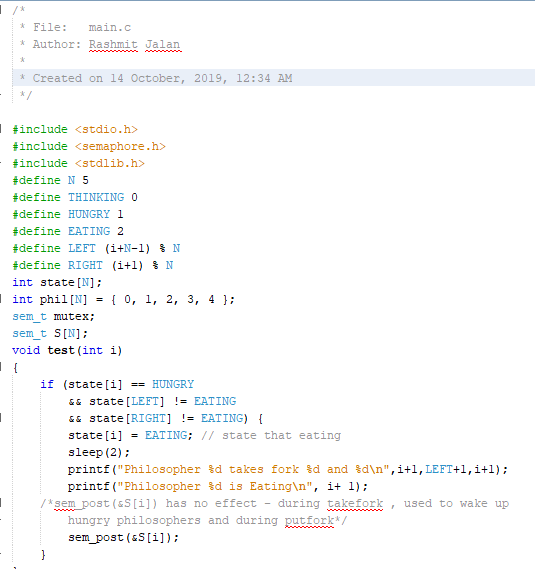
Implement the Dining Philosopher problem using POSIX threads

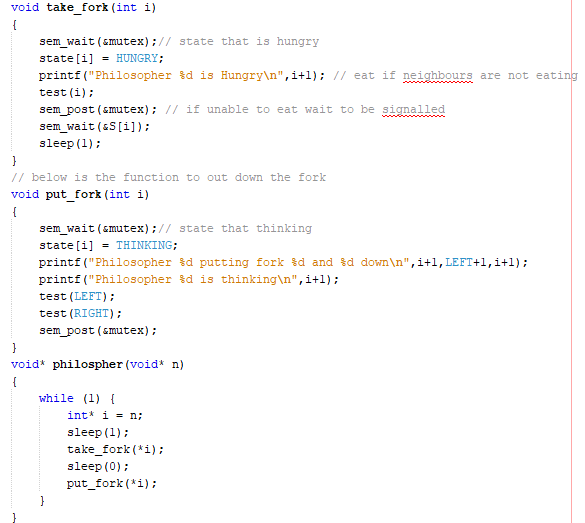
1. Calculations/Computations/Algorithms

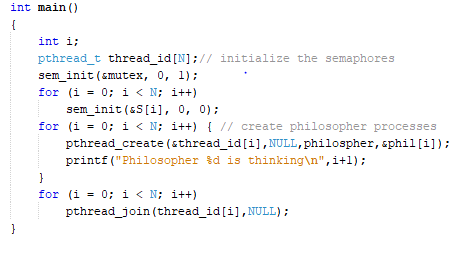
**Algorithm:**

1. Begin
2. Initialize number of philosopher and all the states a philosopher can be in.
3. Declare a function philosopher
4. Begin
5. Change the state of all the Philosopher to thinking
6. Call function take\_fork & put\_fork
7. End
8. Declare Function take\_fork
9. Begin
10. Enter the critical region using semaphore
11. Change the state of philosopher to Hungry
12. Call test function
13. Leave the critical region
14. End
15. Declare function test
16. Begin
17. Check if the philosopher is in hungry state and Philosopher to its left and right are not eating.
18. Change the state of the Philosopher to eating
19. Exit the critical region
20. End
21. Declare function put\_fork
22. Begin
23. If state of philosopher is eating
24. Put mutex semaphore in critical region.
25. Change state of Philosopher to thinking.
26. Call function test with parameters left and right.
27. Leave critical region
28. End
29. Declare main function
30. Begin
31. Initialize semaphore
32. Initialize 5 threads
33. Create a for loop and put every Philosopher into thinking mode.
34. Create 5 threads for each Philosopher
35. Use join function for completion of each thread
36. End

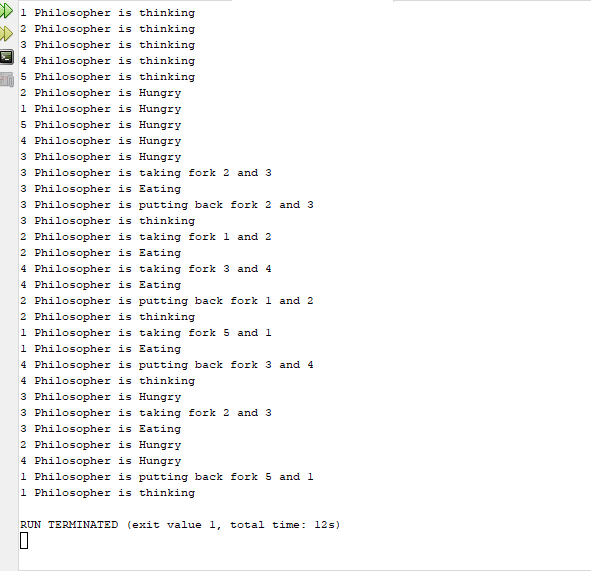
**Program:**







1. **Presentation of Results**



1. **Analysis and Discussions**

We have five different philosophers sitting at a round table with spaghetti bowls in front of them. There are five forks placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks. Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed. The problem is how to design a discipline of behaviour (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. Resource starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example, there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of live lock. If all five philosophers appear in the dining room at exactly the same time and each pick up the left fork at the same time the philosophers will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again.

1. **Conclusions**

The dining and drinking philosopher’s problem are a very old and important problem in the distributed computing field. It was first introduced by Dijkstra and then used by many other researches as a general problem for illustrating mutual exclusion and resource sharing and allocation problem. A lot of algorithms have been introduced to resolve this problem with many options and assumptions which makes each proposed algorithm suitable for specific applications. It was introduced the problem with some of the fundamental and very old solutions for it in the Introduction section. We implemented the problem as a c code. We executed the program and verified it as well.

1. **Comments**

1. **Limitations of Experiments and results**

The Philosophers seem to run in their own threads, and seem to be getting fed. The system isn't locking up as far as I can tell, and no philosopher ever timed out. Again, the fairness can’t be guaranteed by the fact that philosopher sleep random amount of times.

**2. Limitations of Results**

Deadlock in this problem occurs when philosophers put themselves into a position where they can neither think nor eat. Again, the neighbouring philosophers can eat in the same time that is not acceptable.

3**. Learning happened**

Here we have learned how to develop the Dining Philosopher problem using Semaphore. We also learned how to implement the Dining Philosopher problem using POSIX threads. We also learned how to use semaphore and apply appropriate semaphores in different contexts and how to develop concurrent programs using semaphores.

4**. Recommendations**

To get deadlock free solution to the dining – philosopher problem **MONITOR** – based solution to dining philosophers.