K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: A2 **Roll No.: 1911027**

Experiment No. 06

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

TITLE: Implementation of dining philosopher problem using threads.

AIM: Implementation of Process synchronization algorithms using threads – Dining Philosopher problem

Expected Outcome of Experiment:

CO 2. To understand the concept of process, thread and resource management.
CO 3. To understand the concepts of process synchronization and deadlock.

Books/ Journals/ Websites referred:

1. **Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate "Operating Systems" McGraw Hill Third Edition.**
3. **William Stallings, "Operating System Internal & Design Principles", Pearson.**
4. **Andrew S. Tanenbaum, "Modern Operating System", Prentice Hall.**

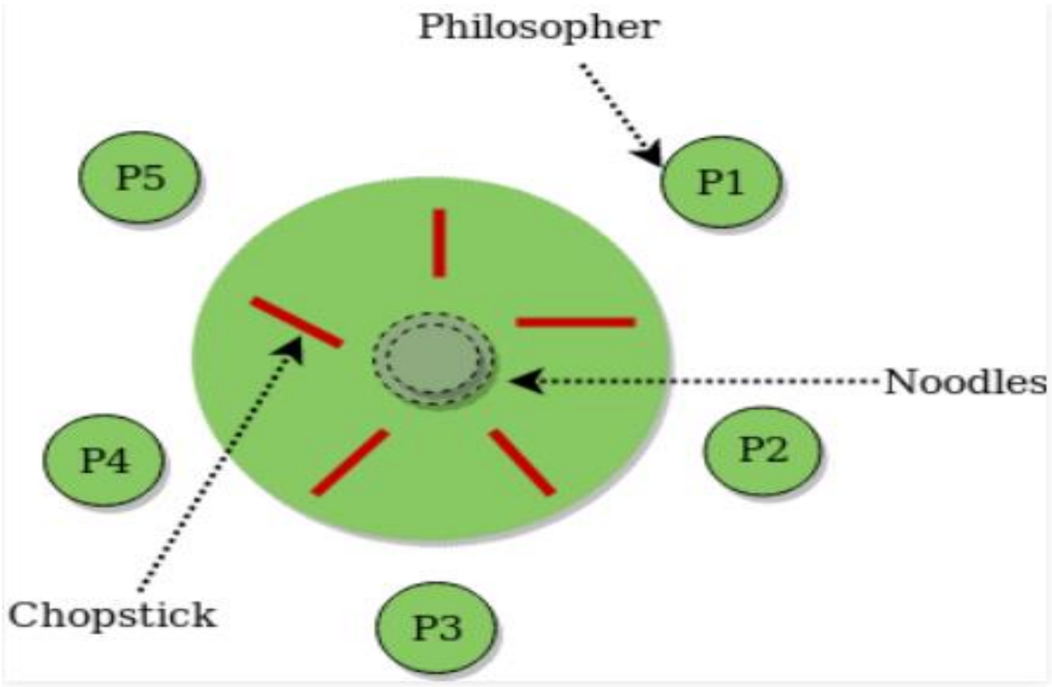
Pre Lab/ Prior Concepts:

Knowledge of Concurrency, Mutual Exclusion, Synchronization, Deadlock, Starvation,threads.

Description of the chosen process synchronization algorithm: Dining Philosopher Problem

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both. The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

Solution of Dining Philosophers Problem: A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore. The structure of the chopstick is shown below − semaphore chopstick [5];



Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher. The structure of a random philosopher i is given as follows –

```
do {
  wait( chopstick[i] );
  wait( chopstick[ (i+1) % 5] );

  . .

  . EATING THE RICE

  .

  signal( chopstick[i] );

  signal( chopstick[ (i+1) % 5] );

  .

  . THINKING

  .

} while(1);
```

In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed. After that, signal operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

Difficulty with the solution: The above solution makes sure that no two neighbouring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs. Some of the ways to avoid deadlock are as follows −

- There should be at most four philosophers on the table.
- An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.

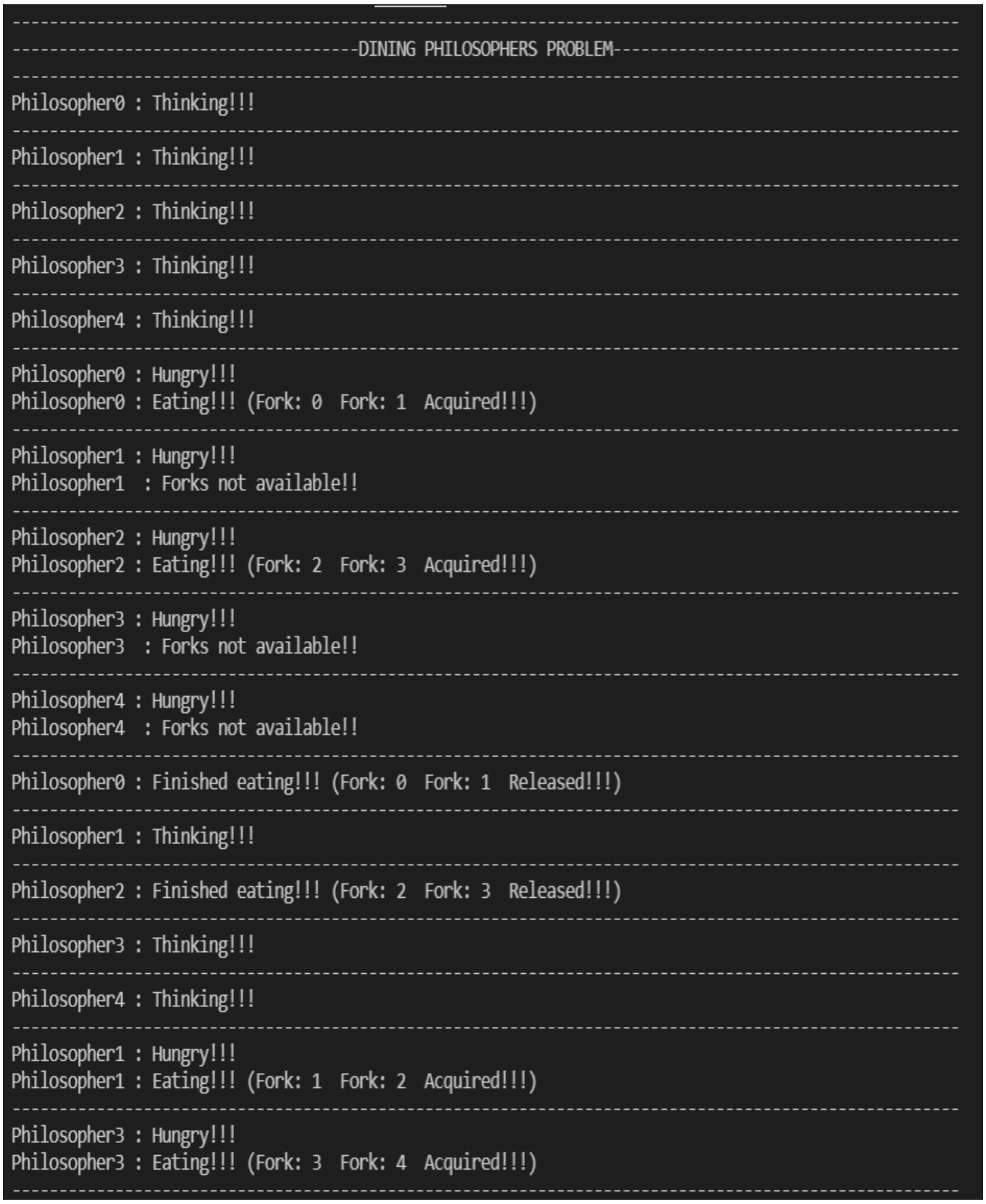
Implementation details: (printout of code)

Code:

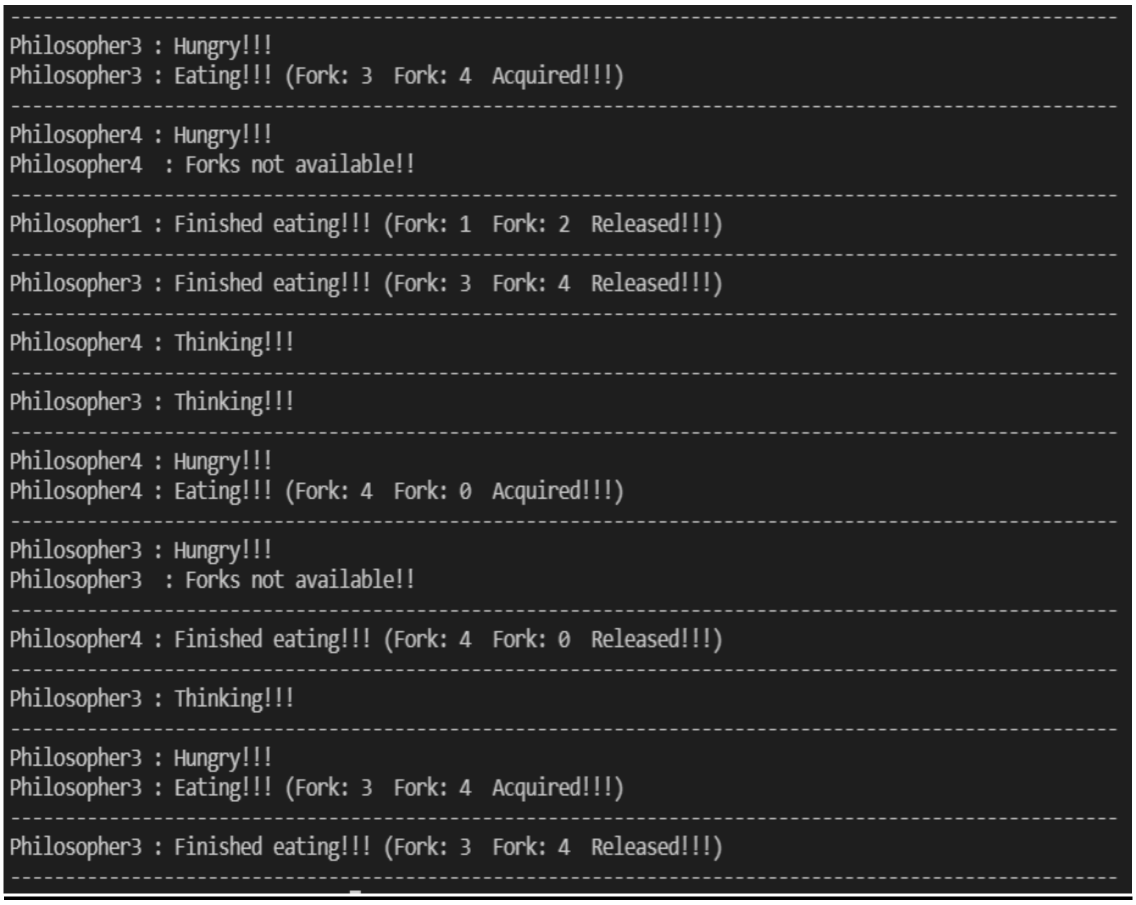
```
import random
import threading
from threading import Thread
import time
def wait(s):
    return s-1

def signal(s):
    return s+1

def simulate():
    num=0
    while(True):
        num=num+1
        print(threading.current_thread().name,": Thinking!!!")
        print("----------------------------------------------------------
-------------------------------------------")
        time.sleep(10)
        print(threading.current_thread().name,": Hungry!!!")
        if(forks[int(threading.current_thread().name[-1])]==1 and
forks[(int(threading.current_thread().name[-1])+1)%n]==1):
            forks[int(threading.current_thread().name[-
1])]=wait(forks[int(threading.current_thread().name[-1])])
            forks[(int(threading.current_thread().name[-
1])+1)%n]=wait(forks[(int(threading.current_thread().name[-1])+1)%n])
            print(threading.current_thread().name,": Eating!!!
(Fork:",int(threading.current_thread().name[-1]),"
Fork:",(int(threading.current_thread().name[-1])+1)%n," Acquired!!!)")
            print("-----------------------------------------------------
-----------------------------------------------")
            time.sleep(10)
            forks[int(threading.current_thread().name[-
1])]=signal(forks[int(threading.current_thread().name[-1])])
            forks[(int(threading.current_thread().name[-
1])+1)%n]=signal(forks[(int(threading.current_thread().name[-1])+1)%n])
```

```
            print(threading.current_thread().name,": Finished eating!!!
(Fork:",int(threading.current_thread().name[-1]),"
Fork:",(int(threading.current_thread().name[-1])+1)%n," Released!!!)")
            print("------------------------------------------------------
-----------------------------------------------")
            time.sleep(10)
            if(random.randint(0,10)>1):
                break
        else:
            print(threading.current_thread().name," : Forks not
available!!")
            print("------------------------------------------------------
-----------------------------------------------")
            time.sleep(10)
            continue

print("------------------------------------------------------------------
-----------------------------------")
print("-------------------------------------DINING PHILOSOPHERS PROBLEM--
-----------------------------------")
print("------------------------------------------------------------------
-----------------------------------")
n=5
forks=[]
for i in range(0,n):
    forks.append(1)
philosophers=[]
for i in range(0,n):
    temp="Philosopher"+str(i)
    philosophers.append(Thread(target=simulate,name=temp))
temp=0
k=0
for i in range(0,len(philosophers)):
    philosophers[k].start()
    k=k+1
    time.sleep(2)
```

Output:



```
------------------------------------------------------------------------------------------------
-----------------------------------DINING PHILOSOPHERS PROBLEM-----------------------------------
------------------------------------------------------------------------------------------------
Philosopher0 : Thinking!!!
------------------------------------------------------------------------------------------------
Philosopher1 : Thinking!!!
------------------------------------------------------------------------------------------------
Philosopher2 : Thinking!!!
------------------------------------------------------------------------------------------------
Philosopher3 : Thinking!!!
------------------------------------------------------------------------------------------------
Philosopher4 : Thinking!!!
------------------------------------------------------------------------------------------------
Philosopher0 : Hungry!!!
Philosopher0 : Eating!!! (Fork: 0  Fork: 1  Acquired!!!)
------------------------------------------------------------------------------------------------
Philosopher1 : Hungry!!!
Philosopher1  : Forks not available!!
------------------------------------------------------------------------------------------------
Philosopher2 : Hungry!!!
Philosopher2 : Eating!!! (Fork: 2  Fork: 3  Acquired!!!)
------------------------------------------------------------------------------------------------
Philosopher3 : Hungry!!!
Philosopher3  : Forks not available!!
------------------------------------------------------------------------------------------------
Philosopher4 : Hungry!!!
Philosopher4  : Forks not available!!
------------------------------------------------------------------------------------------------
Philosopher0 : Finished eating!!! (Fork: 0  Fork: 1  Released!!!)
------------------------------------------------------------------------------------------------
Philosopher1 : Thinking!!!
------------------------------------------------------------------------------------------------
Philosopher2 : Finished eating!!! (Fork: 2  Fork: 3  Released!!!)
------------------------------------------------------------------------------------------------
Philosopher3 : Thinking!!!
------------------------------------------------------------------------------------------------
Philosopher4 : Thinking!!!
------------------------------------------------------------------------------------------------
Philosopher1 : Hungry!!!
Philosopher1 : Eating!!! (Fork: 1  Fork: 2  Acquired!!!)
------------------------------------------------------------------------------------------------
Philosopher3 : Hungry!!!
Philosopher3 : Eating!!! (Fork: 3  Fork: 4  Acquired!!!)
------------------------------------------------------------------------------------------------
```



```
----------------------------------------------------------------------------------------------------
Philosopher3 : Hungry!!!
Philosopher3 : Eating!!! (Fork: 3  Fork: 4  Acquired!!!)
----------------------------------------------------------------------------------------------------
Philosopher4 : Hungry!!!
Philosopher4  : Forks not available!!
----------------------------------------------------------------------------------------------------
Philosopher1 : Finished eating!!! (Fork: 1  Fork: 2  Released!!!)
----------------------------------------------------------------------------------------------------
Philosopher3 : Finished eating!!! (Fork: 3  Fork: 4  Released!!!)
----------------------------------------------------------------------------------------------------
Philosopher4 : Thinking!!!
----------------------------------------------------------------------------------------------------
Philosopher3 : Thinking!!!
----------------------------------------------------------------------------------------------------
Philosopher4 : Hungry!!!
Philosopher4 : Eating!!! (Fork: 4  Fork: 0  Acquired!!!)
----------------------------------------------------------------------------------------------------
Philosopher3 : Hungry!!!
Philosopher3  : Forks not available!!
----------------------------------------------------------------------------------------------------
Philosopher4 : Finished eating!!! (Fork: 4  Fork: 0  Released!!!)
----------------------------------------------------------------------------------------------------
Philosopher3 : Thinking!!!
----------------------------------------------------------------------------------------------------
Philosopher3 : Hungry!!!
Philosopher3 : Eating!!! (Fork: 3  Fork: 4  Acquired!!!)
----------------------------------------------------------------------------------------------------
Philosopher3 : Finished eating!!! (Fork: 3  Fork: 4  Released!!!)
----------------------------------------------------------------------------------------------------
```

Conclusion: Understood the importance of process synchronization in operating system. Also learned the concept of dining philosophers problem and its solution using semaphore. Implemented the solution using python.

Post Lab Descriptive Questions

1.Differentiate between a monitor, semaphore and a binary semaphore?

ANS)

Monitor	Semaphore	Binary Semaphore
1. An abstract data type.	1. An integer variable.	1. A binary variable.
2. Has condition variables.	2. There is no concept of	2. There is no concept of

	condition variables.	condition variables.
3. A process uses procedures to access the shared variable in the monitor.	3. When a process requires to access the semaphore, it performs wait() and signal() operations.	3. When a process requires to access the semaphore, it performs wait() and signal() operations.
4. A synchronization construct that allows threads to have both mutual exclusion and the ability to wait for a certain condition to become true.	4. A variable used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system.	4. It guarantees mutual exclusion, since just one process or thread can enter the critical section at a time.
5. There is no concept of numbering here.	5. A semaphore is a semaphore that has multiple values of the counter. The value can range over an unrestricted domain.	5. A Binary Semaphore is a semaphore whose integer value range over 0 and 1.

2. Define clearly the dining-philosophers problem?
ANS) The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both. Each philosopher is represented by the following pseudocode:

```
process P[i]
 while true do
   {  THINK;
      PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
      EAT;
      PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
   }
```

There are three states of the philosopher: THINKING, HUNGRY, and EATING. Here there are two semaphores: Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time.

The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

3. Identify the scenarios in the dining-philosophers problem that leads to the deadlock situations?

ANS) Deadlock is a situation where no philosophers can eat. This situation arrive when all the philosophers are waiting for chopsticks adjacent to them. Consider a scenario where 4 philosophers are sitting on dining table and there are 4 chopsticks, suppose if all the philosophers are hungry at the same time, say p1 takes chopstick at its left similarly all the philosophers picks chopsticks placed at its left and waiting for its neighbor to release control on its right chopstick. But this will never happen that neighbor will release control so this situation will cause deadlock to occur.

Date: 24 / 11 / 2021 **Signature of faculty in-charge**