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

Batch:	**C2**	**Roll No.:**	**16010121110**

Experiment No. 01

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:

Five philosophers dine together at the same table. Each philosopher has his own plate at the table. There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat his spaghetti when he has both a left and right fork. Thus two forks will only be available when his two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, he will put down both forks. The problem is how to design a regimen (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 (an issue of incomplete information).

Algorithm:

Dijkstra's solution

Source - wikipedia

Dijkstra's solution uses one mutex, one semaphore per philosopher and one state variable per philosopher. This solution is more complex than the resource hierarchy solution.

https://en.wikipedia.org/wiki/Dining_philosophers_problem

Implementation details:

```
import time

import threading

import random

chopSticks = [threading.Lock() for i in range(5+1)]

philosopherStates = [0,0,0,0,0,0]

printer = threading.Lock()

class philosopher():

def __init__(self,name):
```

```
self.name=name

self.state=0

philosopherStates[self.name]=0

"""

0 - Thinking

1 - Hungry

2 - Eating

"""

def run(self):

for i in range(2):

self.think()

self.take_fork()

self.eat()

self.put_fork()
```

```
def think(self):

self.state = 0

philosopherStates[self.name]=0

time.sleep(random.random())

def eat(self):

self.state=2

philosopherStates[self.name]=2

printer.acquire()

print("philosopher ", self.name, "is eating.")

printer.release()

def put_fork(self):

chopSticks[self.name].release()

printer.acquire()

print("philosopher ", self.name, "has dropped up left
stick")
```

```
printer.release()

chopSticks[(self.name+1) %5].release()

printer.acquire()

print("philosopher ", self.name, "has dropped up right
stick")

printer.release()



def take_fork(self):

self.state = 1

philosopherStates[self.name]=1

printer.acquire()

print("philosopher ", self.name, "is hungry.")

printer.release()

while True:
```

```
if(philosopherStates[self.name]!=2 and
philosopherStates[(self.name+1) % 5]!=2 ): # if
philosopher i is hungry and both neighbours are not
eating then eat

chopSticks[self.name].acquire()

chopSticks[(self.name+1) % 5].acquire()

printer.acquire()

print("philosopher ", self.name, "has picked up sticks
")

printer.release()

break



philosopher1=philosopher(1)

philosopher2=philosopher(2)

philosopher3=philosopher(3)

philosopher4=philosopher(4)
```

```
philosopher5=philosopher(5)









T1 = threading.Thread(target =
philosopher1.run,args=())

T2 = threading.Thread(target =
philosopher2.run,args=())

T3 = threading.Thread(target =
philosopher3.run,args=())

T4 = threading.Thread(target =
philosopher4.run,args=())

T5 = threading.Thread(target =
philosopher5.run,args=())

T1.start()

T2.start()

T3.start()
```

```
T4.start()

T5.start()
```

```
philosopher  2 has dropped up left stick
philosopher  2 has dropped up right stick
philosopher  4 is hungry.
philosopher  4 has picked up sticks
philosopher  4 is eating.
philosopher  4 has dropped up left stick
philosopher  4 has dropped up right stick
philosopher  3 is hungry.
philosopher  2 is hungry.
philosopher  2 has picked up sticks
philosopher  2 is eating.
```

Conclusion: Thus we have implemented the dining philosopher problem in Python using multithreading. We have used the Dijkstra's solution to implement the same. This is a solution for threads/processes that need a shared resource.

Post Lab Descriptive Questions

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

A semaphore is a process synchronizing tool. It is basically an integer variable, denoted by "S". The initialization of this variable "S" is done by assigning a number equal to the number of resources present in the system.

There are two functions, wait() and signal(), which are used to modify the value of semaphore "S". The wait() and signal() functions indivisibly change the value of the

semaphore "S". That means, when one process is changing the value of the semaphore "S", another process cannot change the value of the semaphore at the same time.

In operating systems, semaphores are grouped into two categories− counting semaphore and binary semaphore. In a counting semaphore, the value of the semaphore is initialized to the number of resources present in the system. On the other hand, in a binary semaphore, the semaphore "S" has the value "0" or "1"

A binary Semaphore is also called mutex. It can be used for achieving a lock between two processed.

Monitor is also a process synchronization tool. It is an abstract data type that is used for highlevel synchronization of processes. It has been developed to overcome the timing errors that occur while using the semaphore for the process synchronization. Since the monitor is an abstract data type, it contains the shared data variables. These data variables are to be shared by all the processes. Hence, this allows the processes to execute in mutual exclusion.

2. **Define clearly the dining-philosophers problem?**

In computer science, the dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

It was originally formulated in 1965 by Edsger Dijkstra as a student exam exercise, presented in terms of computers competing for access to tape drive peripherals. Soon after, Tony Hoare gave the problem its present form

Five philosophers dine together at the same table. Each philosopher has his own plate at the table. There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat his spaghetti when he has both a left and right fork. Thus two forks will only be available when his two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, he will put down both forks. The problem is how to design a regimen (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 (an issue of incomplete information).

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

All want to eat at same time, so all pick up left chopstick.

Date: 30 Oct 2023 **Signature of faculty in-charge**