AHSANULLAH UNIVERSITY OF SCIENCE AND TECHNOLOGY (AUST)
141 & 142, Love Road, Tejgaon Industrial Area, Dhaka-1208.

Department of Computer Science and Engineering
Program: Bachelor of Science in Computer Science and

Engineering Project

Assignment

Course No : CSE-3213
Course Title : Operating System

Date of Submission :16.02.2025

SubmittedTo : Mr.Md. Moinul Hoque

SubmittedBy :

Name : Abdur Rob Tonim
Student ID : 20210204022

Dining Philosophers Problem :

Definition : The Dining Philosopher Problem, formulated by Edsger Dijkstra, involves five philosophers sitting around a circular table. Each philosopher alternates between thinking and eating. To eat, a philosopher must pick up adjacent fork .

Our challenge :

- If all philosophers pick up one fork at the same time and wait for the other, they will be stuck forever (deadlock).
- If philosophers are not careful, some might starve while others keep eating.

Here we have to avoid deadlocks as well as ensure as much fair resource allocation as possible.

Solution : Here we have the solution code for the Dining Philosophers Problem given below.

```
import threading
import time
import random

class Chopstick:
    def __init__(self):
        self.lock = threading.Lock()

    def pick_up(self):
        self.lock.acquire()
```

```
    def put_down(self):
        self.lock.release()

class Philosopher(threading.Thread):
    def __init__(self, index, left_chopstick, right_chopstick):
        threading.Thread.__init__(self)
        self.index = index
        self.left_chopstick = left_chopstick
        self.right_chopstick = right_chopstick

    def run(self):
        for _ in range(3):
            self.think()
            self.eat()

    def think(self):
        print(f"Philosopher {self.index} is thinking.")
        time.sleep(random.uniform(1, 3))

    def eat(self):
        self.left_chopstick.pick_up()
        self.right_chopstick.pick_up()
        print(f"Philosopher {self.index} is eating.")
        time.sleep(random.uniform(1, 3))
        self.left_chopstick.put_down()
        self.right_chopstick.put_down()

chopsticks = [Chopstick() for _ in range(5)]
philosophers = [Philosopher(i, chopsticks[i], chopsticks[(i+1) % 5]) for i in range(5)]
```

```
for p in philosophers:
    p.start()
for p in philosophers:
    p.join()
```

Different Scenarios :

1. **Deadlock Scenario**: If each philosopher picks up the left chopstick first and waits for the right, a circular wait condition occurs, causing a deadlock. This can be prevented by imposing an ordering on resource allocation.

2. **No Starvation Scenario**: By ensuring a fair locking mechanism, such as allowing only four philosophers to eat at a time, starvation is prevented.

3. **Concurrent Execution Scenario**: The use of locks ensures that multiple philosophers can eat and think concurrently without interference.

4. **Randomized Eating Order**: By introducing random delays, we reduce the likelihood of deadlocks and ensure a more natural execution pattern

.

Sleeping Barber Problem :

Definition : Say we have a barber shop. In the shop we have one barber who cuts hair. A waiting area with a limited number of chairs. Customers who walk in randomly.

Our challenge:

- If there are no customers, the barber sleeps.
- If a customer arrives and the barber is asleep, they wake him up for a haircut.
- If the barber is busy but chairs are available, the customer waits.
- If the shop is full, new customers leave without a haircut.

Here we have to ensure as much fair resource allocation as possible.

Solution : Here we have the solution code for the Sleeping Barber Problem given below :

```
import threading

import time

import random


class BarberShop:

    def __init__(self, num_chairs):

        self.chairs = num_chairs

        self.customers = 0

        self.lock = threading.Semaphore(0)

        self.barber = threading.Thread(target=self.serve_customers)


    def serve_customers(self):

        while True:

            self.lock.acquire()
```

```
        print("Barber is cutting hair.")
        time.sleep(random.uniform(1, 3))
        print("Barber has finished cutting hair.")

    def customer_arrives(self):
        if self.customers < self.chairs:
            self.customers += 1
            print("Customer sits and waits.")
            self.lock.release()
        else:
            print("Customer leaves as no chair is available.")

shop = BarberShop(3)
shop.barber.start()

customers = [threading.Thread(target=shop.customer_arrives) for _ in range(5)]
for c in customers:
    time.sleep(random.uniform(0.5, 2))
    c.start()
for c in customers:
    c.join()
```

Different Scenarios:

1. **Barber Sleeps When No Customers Are Present**: The barber thread waits (blocks) when no customers are present, saving CPU resources.

2. **Customers Leave When Chairs Are Full**: If all chairs are occupied, arriving customers leave, preventing infinite wait conditions.

3. **Synchronization Using Semaphores**: The semaphore ensures customers are served one by one in an orderly manner.

4. **Random Arrival of Customers**: The use of randomized delays simulates real-world customer behavior in a barbershop.