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

Batch: C2	**Roll No.: 16010122323**
Experiment No. 06	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: Implementation of Dining Philosophers problem using mutexes and semaphores.

AIM: Implementation of Process synchronization algorithms using mutexes and semaphore – 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. **Sumitabha Das " UNIX Concepts & Applications", McGraw Hill Second Edition.**

Pre Lab/ Prior Concepts:

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

Description of the chosen process synchronization algorithm:The Dining

Philosophers Problem is a classic problem in the study of process synchronization. It models a scenario where philosophers sit around a circular table with a fork between each pair of philosophers. Each philosopher must pick up two forks (left and right) to eat but can only pick up one fork at a time. The challenge is to ensure that no philosopher starves, and that the system avoids deadlock, where each philosopher waits indefinitely for a fork held by another philosopher. In this implementation, we use semaphores to represent the forks and ensure mutual exclusion. Each philosopher alternates between thinking and eating. When they want to eat, they attempt to acquire the semaphores for the two forks they need. After eating, they release the semaphores, allowing other philosophers to eat.

DINING PHILOSOPHER

Key Elements:

- Philosophers: Represent processes that require resources.
- Forks: Represent shared resources (in this case, the utensils needed for eating).
- Thinking and Eating: Philosophers spend time thinking (non-blocking) and eating (resource-consuming).

Problem Description:

- Each philosopher must pick up the fork to their left and the fork to their right before they can eat.
- After eating, they put down both forks and return to thinking.
- If every philosopher picks up the fork to their left simultaneously, they will all be stuck waiting for the fork to their right, leading to a deadlock.

Synchronization Challenges:

- Deadlock: All philosophers can end up waiting indefinitely if they each hold one fork.
- Starvation: A philosopher might wait indefinitely if the resource allocation doesn’t allow fair access to forks.
- Concurrency: Balancing the access to shared resources without causing significant delays.

Solutions:

- Arbitrator Approach: Introduce a waiter (arbitrator) that allows a philosopher to pick up forks only if both are available, ensuring no deadlock occurs.
- Resource Hierarchy: Enforce an ordering of resource acquisition to prevent circular wait conditions. For example, always pick up the lower-numbered fork first.
- Even/Odd Strategy: Philosophers can follow a rule based on their number (e.g., even-numbered philosophers pick up their left fork first, while odd-numbered ones pick up their right fork first) to reduce contention.

Implementation details:

```
import threading
import time
import random

num_philosophers = 6
forks = [threading.Semaphore(1) for _ in range(num_philosophers)]

def philosopher(index):
    left_cs = forks[index]
    right_cs = forks[(index + 1) % num_philosophers]

    while True:
        print(f"Philosopher {index} is thinking")
        time.sleep(1)

        left_cs.acquire()
        right_cs.acquire()

        print(f"Philosopher {index} is eating")
        time.sleep(1)

        left_cs.release()
        right_cs.release()

if __name__ == "__main__":
```

```
    threads = [threading.Thread(target=philosopher, args=(i,)) for i in
range(num_philosophers)]

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()
```

```
PS D:\Nextcloud\Coding\Google Photos Clone Final> python -u "d:\Nextcloud\Coding\Google Photos Clone Final\tempCodeRunnerFile.python"
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 0 is eating
Philosopher 2 is eating
Philosopher 4 is eating
Philosopher 0 is thinking
Philosopher 2 is thinkingPhilosopher 1 is eating

Philosopher 4 is thinkingPhilosopher 5 is eatingPhilosopher 3 is eating


Philosopher 1 is thinking
Philosopher 3 is thinkingPhilosopher 2 is eatingPhilosopher 5 is thinking

Philosopher 0 is eating
Philosopher 4 is eating

Philosopher 2 is thinkingPhilosopher 0 is thinkingPhilosopher 1 is eating


Philosopher 4 is thinkingPhilosopher 3 is eatingPhilosopher 5 is eating


Philosopher 1 is thinking
Philosopher 5 is thinkingPhilosopher 0 is eatingPhilosopher 3 is thinking
Philosopher 4 is eating
Philosopher 2 is eating


Philosopher 0 is thinking
Philosopher 2 is thinkingPhilosopher 3 is eating
Philosopher 5 is eatingPhilosopher 4 is thinking
Philosopher 1 is eating


Philosopher 3 is thinkingPhilosopher 1 is thinkingPhilosopher 2 is eatingPhilosopher 5 is thinkingPhilosopher 0 is eating

Philosopher 4 is eating


Philosopher 2 is thinking
Philosopher 0 is thinkingPhilosopher 5 is eatingPhilosopher 1 is eating
Philosopher 3 is eatingPhilosopher 4 is thinking



Philosopher 1 is thinking
Philosopher 5 is thinkingPhilosopher 0 is eatingPhilosopher 3 is thinking
Philosopher 2 is eatingPhilosopher 4 is eating


Philosopher 0 is thinkingPhilosopher 4 is thinkingPhilosopher 3 is eatingPhilosopher 5 is eatingPhilosopher 2 is thinking

Philosopher 1 is eating



Philosopher 3 is thinkingPhilosopher 1 is thinkingPhilosopher 2 is eatingPhilosopher 5 is thinking
Philosopher 4 is eating
```

Used mutex:

```
import threading
```

```
import time

num_philosophers = 2
```

```
# Create a mutex for each fork (chopstick)
forks = [threading.Lock() for _ in range(num_philosophers)]


def philosopher(index):
    left_cs = forks[index]
    right_cs = forks[(index + 1) % num_philosophers]

    while True:
        print(f"Philosopher {index} is thinking")
        time.sleep(1)  # Simulate thinking

        with left_cs:  # Lock the left chopstick
            with right_cs:  # Lock the right chopstick
                print(f"Philosopher {index} is eating")
                time.sleep(1)  # Simulate eating


if __name__ == "__main__":
    threads = [
        threading.Thread(target=philosopher, args=(i,)) for i in
range(num_philosophers)
    ]

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()
```

```
PS D:\Nextcloud\Coding\Google Photos Clone Final> python -u "d:\Nextcloud\Coding\Google Photos Clone Final\tempCodeRunnerFile.python"
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 0 is eating
Philosopher 0 is thinking
Philosopher 1 is eating
Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinking
Philosopher 0 is eating
Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinking
Philosopher 1 is eating
Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinking
Philosopher 1 is eating
Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinking
Philosopher 1 is eating
Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinking
Philosopher 0 is eating
Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinking
Philosopher 1 is eating
Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinking
Philosopher 0 is eating
Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating

Philosopher 0 is thinkingPhilosopher 1 is eating

Philosopher 1 is thinkingPhilosopher 0 is eating
```

Conclusion:

Learnt dining philosophers using mutexes and semaphores.

Post Lab Descriptive Questions

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

Monitor

- High-level synchronization construct that combines mutual exclusion and condition synchronization.
- Only one thread can execute a monitor's method at a time.

Semaphore

- Low-level synchronization primitive using a counter to manage access to shared resources.
- Can be counting (multiple accesses) or binary (0 or 1).

Binary Semaphore

- A type of semaphore that only has two states (0 or 1).
- Ensures mutual exclusion, allowing only one thread to access a resource at a time.

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

- Circular Wait : Each philosopher picks up one fork (resource) and waits for the other fork. If all five philosophers do this simultaneously, they will each hold one fork and wait for the other, creating a circular dependency.
- Hold and Wait : If a philosopher picks up one fork and then waits for the second fork while holding the first, this can lead to multiple philosophers holding one fork each and waiting indefinitely for the second fork.
- No Preemption : Once a philosopher has picked up a fork, they cannot put it down until they have both forks. This lack of preemption means that resources cannot be reclaimed, which contributes to the deadlock.

- All Philosophers Act Simultaneously:If all philosophers try to pick up forks at the same time, they can all end up holding one fork and waiting for another, which creates a deadlock scenario.

Deadlock Conditions

- Mutual Exclusion: Forks cannot be shared.
- Hold and Wait: Philosophers can hold one fork while waiting for another.
- No Preemption: Philosophers cannot forcibly take a fork from another.
- Circular Wait: There exists a circular chain of philosophers each waiting for a fork held by the next.

3. Which of the following can be used to avoid deadlock in the Dining Philosophers Problem?
a. Using a semaphore initialized to the number of philosophers.
b. **Using a semaphore initialized to one less than the number of philosophers.**
c. Using a mutex for each philosopher.
d. Using a monitor for each fork
4. Which synchronization construct encapsulates shared variables, synchronization primitives, and operations on shared variables?
a. Semaphore
b. Binary Semaphore
c. Monitor
d. Mutex

Date: _____ **Signature of faculty in-charge**