S.B. JAIN INSTITUTE OF TECHNOLOGY MANAGEMENT & RESEARCH, NAGPUR

Practical 06

Aim: Considered there are N 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.

Write a program to solve the problem using process synchronization technique.

Name: Sayali Shamrao Misal

USN: CM25D007

Semester/Year: IV/II

Academic Session: 2025-2026

Date of Performance: 17/02/2026

Date of Submission:

Aim: Considered there are N 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.

Write a program to solve the problem using process synchronization technique.

❖ **Objectives:**
  a) T**o implement process synchronization** using semaphores or mutex locks to prevent deadlock and ensure philosophers can eat without conflicts.
  b) **To apply the Dining Philosophers Problem** as a classic example of synchronization in concurrent programming.
  c) **To ensure no two adjacent philosophers** pick up the same chopstick simultaneously, avoiding resource contention.
  d) **To demonstrate deadlock prevention** and starvation avoidance using techniques like ordering of resource allocation or introducing an arbitrator.

❖ **Requirements:**
  ✓ **Hardware Requirements:**

  • Processor: Minimum 1 GHz
  • RAM: 512 MB or higher
  • Storage: 100 MB free space

  ✓ **Software Requirements:**

  • Operating System: Linux/Unix-based
  • Shell: Bash 4.0 or higher
  • Text Editor: Nano, Vim, or any preferred editor

❖ **Theory:**

The Dining Philosopher Problem states that K philosophers are 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.

Semaphore Solution to Dining Philosopher

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 put it down 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. **Outline of a philosopher process:**

Var successful: boolean;

```
repeat
    successful:= false; while (not
    successful)
    if both forks are available then lift the
    forks one at a time; successful:= true;
   if successful = false then
   block(Pi);
   {eat}
   put down both forks;
   if left neighbor is waiting for his right fork then
   activate (left neighbor);
    if right neighbor is waiting for his left fork then
   activate( right neighbor);
  {think} forever
```

The steps for the Dining Philosopher Problem solution using semaphores are as follows:
1. Initialize the semaphores for each fork to 1 (indicating that they are available).
2.   Initialize a binary semaphore (mutex) to 1 to ensure that only one philosopher can attempt to pick up a fork at a time.
3.   For each philosopher process, create a separate thread that executes the following code:
- **While true:**
o Think for a random amount of time.

- Acquire the mutex semaphore to ensure that only one philosopher can attempt to pick up a fork at a time.
  - Attempt to acquire the semaphore for the fork to the left.
- If successful, attempt to acquire the semaphore for the fork to the right.
- If both forks are acquired successfully, eat for a random amount of time and then release both semaphores.
- If not successful in acquiring both forks, release the semaphore for the fork to the left (if acquired) and then release the mutex semaphore and go back to thinking.

  4. Run the philosopher threads concurrently.

  By using semaphores to control access to the forks, the Dining Philosopher Problem can be solved in a way that avoids deadlock and starvation. The use of the mutex semaphore ensures that only one philosopher can attempt to pick up a fork at a time, while the use of the fork semaphores ensures that a philosopher can only eat if both forks are available.

  Overall, the Dining Philosopher Problem solution using semaphores is a classic example of how synchronization mechanisms can be used to solve complex synchronization problems in concurrent programming.

  As you can see from the output, each philosopher takes turns thinking and eating, and there is no deadlock or starvation.

  Problem with Dining Philosopher

  We have demonstrated that no two nearby philosophers can eat at the same time from the aforementioned solution to the dining philosopher problem. The problem with the above solution is that it might result in a deadlock situation. If every philosopher picks their left chopstick simultaneously, a deadlock results, and no philosopher can eat. This situation occurs when this happens.

  Some of the solutions include the following:

1. The maximum number of philosophers at the table should not exceed four; in this case, philosopher P3 will have access to chopstick C4; he will then begin eating, and when he is finished, he will put down both chopsticks C3 and C4; as a result, semaphore C3 and C4 will now be incremented to 1.
2. Now that philosopher P2, who was holding chopstick C2, will also have chopstick C3, he will do the same when finished eating, allowing other philosophers to eat.
3. While a philosopher in an odd position should select the right chopstick first, a philosopher in an even position should select the left chopstick and then the right chopstick.
4. A philosopher should only be permitted to choose their chopsticks if both of the available chopsticks (the left and the right) are available at the same time.
5. All four of the initial philosophers (P0, P1, P2, and P3) should choose the left chopstick before choosing the right, while P4 should choose the right chopstick

start eating and will put down both chopsticks once finishes and let others eat which removes the problem of deadlock.

❖ **CODE:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define N 5

sem_t chopstick[N];
pthread_t philosopher[N];

void* dine(void* arg)
{
    int id = *(int*)arg;
    int left = id;
    int right = (id + 1) % N;

    while (1)
    {
        printf("Philosopher %d is thinking\n", id);
        sleep(1);

        // Pick up left chopstick
        sem_wait(&chopstick[left]);
        printf("Philosopher %d picked up left chopstick %d\n", id, left);

        sem_wait(&chopstick[right]);
        printf("Philosopher %d picked up right chopstick %d\n", id, right);

        // Eat
        printf("Philosopher %d is eating\n", id);
        sleep(2);

        sem_post(&chopstick[left]);

        sem_post(&chopstick[right]);

        printf("Philosopher %d put down chopsticks and starts thinking\n", id);
    }
}
int main()
{
    int i;
    int ids[N];

    // Initialize semaphores
    for (i = 0; i < N; i++)
        sem_init(&chopstick[i], 0, 1);

    // Create philosopher threads
    for (i = 0; i < N; i++)
    {
        ids[i] = i;

        pthread_create(&philosopher[i], NULL, dine, &ids[i]);
    }

    // Join threads
    for (i = 0; i < N; i++)

        pthread_join(philosopher[i], NULL);

    return 0;
}
```

❖ **OUTPUT:**

```
sayali@DESKTOP-T8KMP91: ~        ×        +    ∨

sayali@DESKTOP-T8KMP91:~$ gcc dining.c -o dining -lpthrea
sayali@DESKTOP-T8KMP91:~$ ./dining
=== DINING PHILOSOPHERS SIMULATION ===

[Round 1] Philosopher 0 THINKING

[Round 1] Philosopher 1 THINKING

[Round 1] Philosopher 2 THINKING

[Round 1] Philosopher 3 THINKING

[Round 1] Philosopher 4 THINKING
[Round 1] Philosopher 0 PICKED LEFT 0
[Round 1] Philosopher 1 PICKED LEFT 1
[Round 1] Philosopher 1 PICKED RIGHT 2
[Round 1] Philosopher 1 EATING
[Round 1] Philosopher 3 PICKED LEFT 3
[Round 1] Philosopher 4 PICKED LEFT 4
[Round 1] Philosopher 1 RELEASED chopsticks
[Round 1] Philosopher 2 PICKED LEFT 2
[Round 1] Philosopher 0 PICKED RIGHT 1
[Round 1] Philosopher 0 EATING

[Round 2] Philosopher 1 THINKING
[Round 1] Philosopher 0 RELEASED chopsticks

[Round 2] Philosopher 0 THINKING
[Round 1] Philosopher 4 PICKED RIGHT 0
[Round 1] Philosopher 4 EATING
[Round 2] Philosopher 1 PICKED LEFT 1
[Round 1] Philosopher 4 RELEASED chopsticks

[Round 2] Philosopher 4 THINKING
[Round 1] Philosopher 3 PICKED RIGHT 4
[Round 1] Philosopher 3 EATING
[Round 2] Philosopher 0 PICKED LEFT 0
[Round 1] Philosopher 3 RELEASED chopsticks

[Round 2] Philosopher 3 THINKING
[Round 2] Philosopher 4 PICKED LEFT 4
[Round 1] Philosopher 2 PICKED RIGHT 3
[Round 1] Philosopher 2 EATING
[Round 1] Philosopher 2 RELEASED chopsticks

[Round 2] Philosopher 2 THINKING
[Round 2] Philosopher 1 PICKED RIGHT 2
[Round 2] Philosopher 1 EATING
[Round 2] Philosopher 3 PICKED LEFT 3
[Round 2] Philosopher 1 RELEASED chopsticks

[Round 3] Philosopher 1 THINKING
[Round 2] Philosopher 0 PICKED RIGHT 1
[Round 2] Philosopher 2 PICKED LEFT 2
[Round 2] Philosopher 0 EATING
[Round 2] Philosopher 0 RELEASED chopsticks
[Round 2] Philosopher 4 PICKED RIGHT 0
```

```
[Round 2] Philosopher 0 RELEASED chopsticks
[Round 2] Philosopher 4 PICKED RIGHT 0
[Round 2] Philosopher 4 EATING
[Round 3] Philosopher 1 PICKED LEFT 1

[Round 3] Philosopher 0 THINKING
[Round 2] Philosopher 4 RELEASED chopsticks

[Round 3] Philosopher 4 THINKING
[Round 2] Philosopher 3 PICKED RIGHT 4
[Round 2] Philosopher 3 EATING
[Round 3] Philosopher 0 PICKED LEFT 0
[Round 2] Philosopher 3 RELEASED chopsticks

[Round 3] Philosopher 3 THINKING
[Round 2] Philosopher 2 PICKED RIGHT 3
[Round 2] Philosopher 2 EATING
[Round 3] Philosopher 4 PICKED LEFT 4
[Round 2] Philosopher 2 RELEASED chopsticks

[Round 3] Philosopher 2 THINKING
[Round 3] Philosopher 1 PICKED RIGHT 2
[Round 3] Philosopher 1 EATING
[Round 3] Philosopher 3 PICKED LEFT 3
[Round 3] Philosopher 1 RELEASED chopsticks
Philosopher 1 finished all rounds.
[Round 3] Philosopher 0 PICKED RIGHT 1
[Round 3] Philosopher 0 EATING
[Round 3] Philosopher 2 PICKED LEFT 2
[Round 3] Philosopher 0 RELEASED chopsticks
Philosopher 0 finished all rounds.
[Round 3] Philosopher 4 PICKED RIGHT 0
[Round 3] Philosopher 4 EATING
[Round 3] Philosopher 4 RELEASED chopsticks
Philosopher 4 finished all rounds.
[Round 3] Philosopher 3 PICKED RIGHT 4
[Round 3] Philosopher 3 EATING
[Round 3] Philosopher 3 RELEASED chopsticks
Philosopher 3 finished all rounds.
[Round 3] Philosopher 2 PICKED RIGHT 3
[Round 3] Philosopher 2 EATING
[Round 3] Philosopher 2 RELEASED chopsticks
Philosopher 2 finished all rounds.

=== SIMULATION COMPLETED ===
sayali@DESKTOP-T8KMP91:~$ nano sjf.c
```

❖ **Conclusion**: In this practical, we conclude that process synchronization techniques, such as semaphores and mutex locks, effectively manage shared resources in concurrent systems. By implementing the Dining Philosophers Problem, we ensured deadlock prevention, starvation avoidance, and efficient resource allocation, demonstrating the importance of synchronization in multi-process environments.

❖ **Discussion Questions:**

1. **What is the main problem in the Dining Philosophers Problem?**

2. **Which synchronization techniques can be used to solve this problem?**

3. **How can deadlock be avoided in this problem?**

4. **Why is starvation a concern in the Dining Philosophers Problem?**

5. **What is the role of semaphores in solving this problem?**

❖ **References:**
   https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/

\

_____

Date:___/___/2026

Signature
Course Coordinator
B.Tech CSE(AIML)
Sem: 4 / 2025-26