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: Tarik A. Ansari

USN: CM24054

Semester / Year:

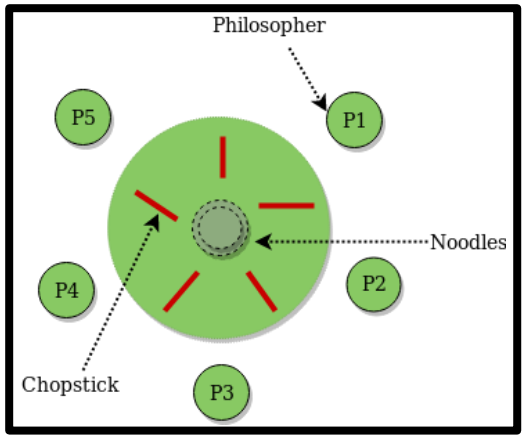
Academic Session:

Date of Performance:

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:**
  - 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.

Implementation of the Dining Philosopher Problem solution using semaphores in Python

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 5             // number of philosophers
#define THINKING 0
#define HUNGRY   1
#define EATING   2

sem_t chopsticks[N];      // one semaphore per chopstick
pthread_t philosophers[N];

int philosopher_id[N];

void think(int id) {
   printf("Philosopher %d is thinking...\n", id);
   sleep(rand() % 3 + 1);  // simulate thinking for 1-3 seconds
}

void eat(int id) {
   printf("Philosopher %d is EATING.\n", id);
   sleep(rand() % 2 + 1);  // simulate eating for 1-2 seconds
}

void* philosopher_life(void* arg) {
   int id = *(int*)arg;
   int left = id;             // left chopstick index
   int right = (id + 1) % N;    // right chopstick index

   while (1) {
      think(id);

      // Asymmetric pickup order to avoid deadlock
      if (id % 2 == 0) {
         // Even: pick left first, then right
         sem_wait(&chopsticks[left]);
         printf("Philosopher %d picked up left chopstick %d\n", id, left);
         sem_wait(&chopsticks[right]);
         printf("Philosopher %d picked up right chopstick %d\n", id, right);
      }
```

```
            else {
            // Odd: pick right first, then left
            sem_wait(&chopsticks[right]);
            printf("Philosopher %d picked up right chopstick %d\n", id, right);
            sem_wait(&chopsticks[left]);
            printf("Philosopher %d picked up left chopstick %d\n", id, left);
        }

        eat(id);

        // Put down both chopsticks
        sem_post(&chopsticks[left]);
        sem_post(&chopsticks[right]);
        printf("Philosopher %d put down chopsticks\n", id);
    }
    return NULL;
}

int main() {
    srand(time(NULL));

    // Initialize semaphores for chopsticks
    for (int i = 0; i < N; i++) {
        sem_init(&chopsticks[i], 0, 1);
        philosopher_id[i] = i;
    }

    // Create philosopher threads
    for (int i = 0; i < N; i++) {
        pthread_create(&philosophers[i], NULL, philosopher_life,
&philosopher_id[i]);
    }

    // Wait for threads (they run forever, but we keep main alive)
    for (int i = 0; i < N; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Cleanup (never reached in this example)
    for (int i = 0; i < N; i++) {
        sem_destroy(&chopsticks[i]);
    }
    return 0;
}
```

OUTPUT:

```
ubuntu@ubuntu:~$ nano dining_philosopher.c
ubuntu@ubuntu:~$ gcc -o dining_philosopher dining_philosopher.c -lpthread
ubuntu@ubuntu:~$ ./dining_philosopher
Philosopher 0 is thinking...
Philosopher 1 is thinking...
Philosopher 4 is thinking...
Philosopher 2 is thinking...
Philosopher 3 is thinking...
Philosopher 4 picked up left chopstick 4
Philosopher 4 picked up right chopstick 0
Philosopher 4 is EATING.
Philosopher 1 picked up right chopstick 2
Philosopher 1 picked up left chopstick 1
Philosopher 1 is EATING.
Philosopher 4 put down chopsticks
Philosopher 3 picked up right chopstick 4
Philosopher 0 picked up left chopstick 0
Philosopher 4 is thinking...
Philosopher 3 picked up left chopstick 3
Philosopher 3 is EATING.
Philosopher 3 put down chopsticks
Philosopher 3 is thinking...
Philosopher 1 put down chopsticks
Philosopher 1 is thinking...
Philosopher 0 picked up right chopstick 1
Philosopher 0 is EATING.
Philosopher 2 picked up left chopstick 2
Philosopher 2 picked up right chopstick 3
Philosopher 2 is EATING.
Philosopher 3 picked up right chopstick 4
```

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.

   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 before choosing the left. As a result, P4 will be forced to hold his right chopstick first because his right chopstick, C0, is already being held by philosopher P0 and its value is set to 0. As a result, P4 will become trapped in an infinite loop and chopstick C4 will remain empty. As a result, philosopher P3 has both the left C3 and the right C4. chopstick available, therefore it will start eating and will put down both chopsticks once finishes and let others eat which removes the problem of deadlock.

❖ **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