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: Pranav Mohan Paunikar
USN: CM24032
Semester / Year: III/II
Academic Session:2025-26
Date of Performance:13/1/26
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:

- To implement process synchronization using semaphores or mutex locks to prevent deadlock and ensure philosophers can eat without conflicts.
- To apply the Dining Philosophers Problem as a classic example of synchronization in concurrent programming.
- To ensure no two adjacent philosophers pick up the same chopstick simultaneously, avoiding resource contention.
- 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 illustrates synchronization challenges in concurrent programming. **K philosophers** are seated around a circular table with one chopstick placed between each pair of philosophers. Each philosopher requires **two adjacent chopsticks** to eat—the one on their left and the one on their right. A chopstick can only be used by one philosopher at a time, creating potential competition for resources.

1. Philosopher Process 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])
}
```

2. States and Synchronization Mechanisms

Each philosopher exists in one of three states:

- **THINKING**: Contemplating, not requiring chopsticks
- **HUNGRY**: Needing both chopsticks to eat
- **EATING**: Successfully acquired both chopsticks

The solution employs two types of semaphores:

- **Mutex**: A binary semaphore ensuring mutual exclusion—only one philosopher can access the pickup/putdown operations at a time
- **Semaphore array**: Controls individual philosopher behavior and prevents race conditions

3. Detailed Philosopher Process Outline

```
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(P[i]);

   {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
```

4. Implementation Steps

1. **Initialize semaphores** for each fork to 1 (indicating availability)
2. **Initialize a binary semaphore (mutex)** to 1, ensuring only one philosopher attempts to pick up forks at any given time
3. **Create philosopher threads** executing the following concurrent process:

```
While true:
   Think for a random amount of time
   Acquire mutex semaphore (ensures exclusive access)
   Attempt to acquire left fork semaphore
   If successful, attempt to acquire right fork semaphore

   If both forks acquired:
      Eat for a random amount of time
      Release both fork semaphores
   Else:
      Release left fork semaphore (if acquired)
```

Release mutex semaphore
Return to thinking

Run philosopher threads concurrently

5. Deadlock Problem

The above solution may result in a deadlock situation. If every philosopher picks their left chopstick simultaneously, a deadlock results, and no philosopher can eat.

6. Deadlock Solutions

Solution 1: Limit Philosophers

The maximum number of philosophers at the table should not exceed four. In this case:

- Philosopher P3 will have access to chopstick C4 and begin eating
- After finishing, P3 puts down both chopsticks C3 and C4, incrementing semaphores C3 and C4 to 1
- Philosopher P2, holding chopstick C2, can now acquire chopstick C3 and eat
- This chain allows other philosophers to eat sequentially

Solution 2: Even/Odd Acquisition Pattern

A philosopher in an even position should select the left chopstick first, then the right chopstick. A philosopher in an odd position should select the right chopstick first, then the left chopstick.

Solution 3: Simultaneous Acquisition

A philosopher should only be permitted to pick up chopsticks if both the left and right chopsticks are available at the same time.

Solution 4: Modified Order for One Philosopher

All four initial philosophers (P0, P1, P2, P3) choose the left chopstick before the right, while P4 chooses the right chopstick before the left:

- P4 attempts to hold right chopstick C0 first, but C0 is already held by philosopher P0
- P4 becomes trapped waiting, leaving chopstick C4 empty
- Philosopher P3 now has both left C3 and right C4 chopsticks available and begins eating
- After finishing, P3 puts down both chopsticks, allowing others to eat
- This removes the deadlock problem

8. Advantages of the Semaphore Solution

This approach effectively addresses the Dining Philosopher Problem by:

- **Preventing deadlock** through controlled resource acquisition
- **Avoiding starvation** by checking and activating waiting neighbors
- **Ensuring mutual exclusion** with mutex semaphores during critical sections
- **Maintaining concurrency** while protecting shared resources

CODE:

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

#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT (philosopher_number + N - 1) % N
#define RIGHT (philosopher_number + 1) % N

pthread_mutex_t mutex;
pthread_cond_t cond[N];
int state[N];

void test(int philosopher_number) {
   if (state[philosopher_number] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {

      state[philosopher_number] = EATING;
      printf("Philosopher %d starts EATING\n", philosopher_number + 1);
      pthread_cond_signal(&cond[philosopher_number]);
   }
}

void take_chopsticks(int philosopher_number) {
   pthread_mutex_lock(&mutex);

   state[philosopher_number] = HUNGRY;
   printf("Philosopher %d is HUNGRY\n", philosopher_number + 1);

   test(philosopher_number);

   while (state[philosopher_number] != EATING) {
      pthread_cond_wait(&cond[philosopher_number], &mutex);
   }

   pthread_mutex_unlock(&mutex);
}

void put_chopsticks(int philosopher_number) {
   pthread_mutex_lock(&mutex);

   state[philosopher_number] = THINKING;
   printf("Philosopher %d finishes EATING and starts THINKING\n", philosopher_number + 1);
```

```
    test(LEFT);
    test(RIGHT);

    pthread_mutex_unlock(&mutex);
}

void* philosopher(void* num) {
    int philosopher_number = *(int*)num;

    while (1) {
        printf("Philosopher %d is THINKING\n", philosopher_number + 1);
        sleep(rand() % 3 + 1);

        take_chopsticks(philosopher_number);

        sleep(rand() % 3 + 1); // Eating

        put_chopsticks(philosopher_number);
    }
}

int main() {
    pthread_t thread_id[N];
    int phil[N] = {0, 1, 2, 3, 4};

    pthread_mutex_init(&mutex, NULL);
    for (int i = 0; i < N; i++) {
        pthread_cond_init(&cond[i], NULL);
        state[i] = THINKING;
    }

    for (int i = 0; i < N; i++) {
        pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
        printf("Philosopher %d is seated\n", i + 1);
    }

    for (int i = 0; i < N; i++) {
        pthread_join(thread_id[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    for (int i = 0; i < N; i++) {
        pthread_cond_destroy(&cond[i]);
    }

    return 0;
}
```

OUTPUT:

```
Philosopher 2 finishes EATING and starts THINKING
Philosopher 2 is THINKING
Philosopher 4 finishes EATING and starts THINKING
Philosopher 2 is HUNGRY
Philosopher 2 starts EATING
Philosopher 4 is THINKING
Philosopher 1 is HUNGRY
Philosopher 5 is HUNGRY
Philosopher 5 starts EATING
Philosopher 3 is HUNGRY
Philosopher 4 is HUNGRY
Philosopher 5 finishes EATING and starts THINKING
Philosopher 4 starts EATING
Philosopher 5 is THINKING
Philosopher 2 finishes EATING and starts THINKING
Philosopher 1 starts EATING
Philosopher 2 is THINKING
Philosopher 4 finishes EATING and starts THINKING
Philosopher 3 starts EATING
Philosopher 4 is THINKING
Philosopher 2 is HUNGRY
Philosopher 5 is HUNGRY
Philosopher 1 finishes EATING and starts THINKING
Philosopher 5 starts EATING
Philosopher 1 is THINKING
Philosopher 4 is HUNGRY
Philosopher 3 finishes EATING and starts THINKING
Philosopher 2 starts EATING
Philosopher 3 is THINKING
Philosopher 5 finishes EATING and starts THINKING
Philosopher 4 starts EATING
Philosopher 5 is THINKING
Philosopher 3 is HUNGRY
Philosopher 2 finishes EATING and starts THINKING
Philosopher 2 is THINKING
Philosopher 1 is HUNGRY
Philosopher 1 starts EATING
Philosopher 5 is HUNGRY
```

1.

Conclusion: Through this practical, I learned how **semaphores and mutex locks** help manage shared resources in concurrent programming. The Dining Philosophers Problem taught me that without proper synchronization, processes can deadlock or starve. By implementing these techniques, I successfully prevented conflicts and ensured all philosophers got a fair chance to eat—demonstrating the real-world importance of process synchronization.

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