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: Yashaswini Kalambe

USN:CD24063

Semester / Year: 4th SEM / 2nd YEAR

Academic Session: 2025-26

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.

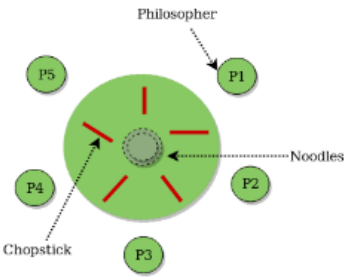


Fig: 1. Dining Philosopher Problem

❖ 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:

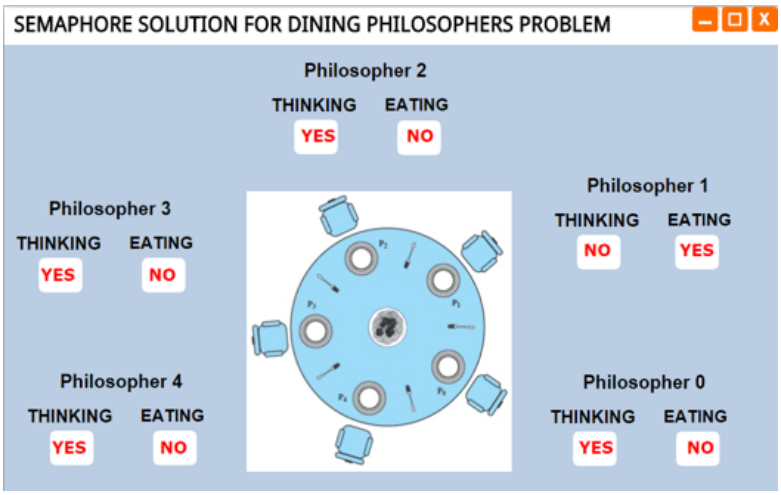
✓ Hardware Requirements:

- 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.

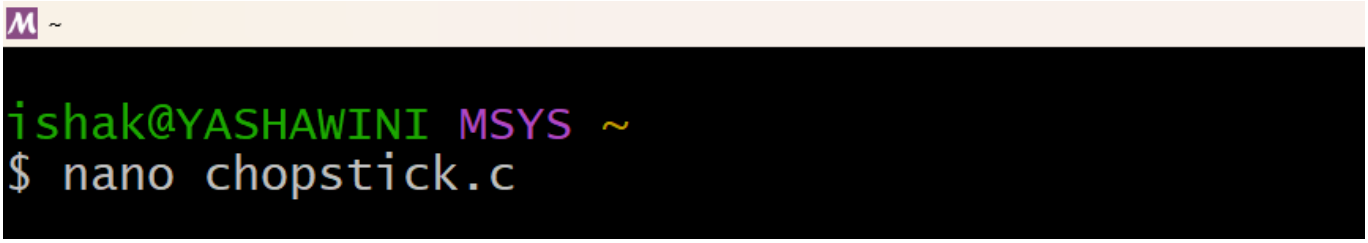
4. Run the philosopher threads concurrently.

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:



CODE:

```
  GNU nano 8.7                                  chopstick.c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 5

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

void* eat(void* arg)
{
    int id = *(int*)arg;

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

    sem_wait(&chopstick[id]);               // Pick left chopstick
    sem_wait(&chopstick[(id + 1) % N]);     // Pick right chopstick

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

    sem_post(&chopstick[id]);               // Put left chopstick
    sem_post(&chopstick[(id + 1) % N]);     // Put right chopstick

    printf("Philosopher %d finished eating\n", id);
    return NULL;
}

int main()
{
    int i, id[N];

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

    for (i = 0; i < N; i++) {
        id[i] = i;
        pthread_create(&philosopher[i], NULL, eat, &id[i]);
    }

    for (i = 0; i < N; i++)

^G Help        ^O Write Out   ^F Where Is    ^K Cut         ^T Execute
^X Exit        ^R Read File   ^\ Replace     ^U Paste       ^J Justify
```

OUTPUT:

```
~

ishak@YASHAWINI MSYS ~
$ nano chopstick.c

ishak@YASHAWINI MSYS ~
$ gcc chopstick.c -o chopstick && ./chopstick
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 2 is eating
Philosopher 4 is eating
Philosopher 2 finished eating
Philosopher 1 is eating
Philosopher 4 finished eating
Philosopher 3 is eating
Philosopher 3 finished eating
Philosopher 1 finished eating
Philosopher 0 is eating
Philosopher 0 finished eating
```

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 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?

Ans: The main problem is ensuring proper synchronization among philosophers who share limited resources (chopsticks) to avoid deadlock and starvation. Each philosopher needs two adjacent chopsticks to eat, but if all philosophers pick up one chopstick simultaneously and wait for the second, a deadlock can occur. The challenge is to design a solution where philosophers can eat without causing system-wide blocking.

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

Ans: Various synchronization techniques can be used, such as mutex locks, semaphores, and monitors. Semaphores can be used to control access to chopsticks, ensuring only one philosopher picks up a chopstick at a time. Monitors provide a structured way to manage concurrency by encapsulating shared resources and conditions for safe execution. Mutex locks ensure that only one philosopher can access a chopstick at a time, preventing race conditions.

3. How can deadlock be avoided in this problem?

Ans: Deadlock can be avoided by implementing strategies such as ensuring that at most (N-1) philosophers pick up chopsticks at a time, enforcing an ordered resource allocation (e.g., odd-numbered philosophers pick left chopstick first while even-numbered philosophers pick right chopstick first), or introducing an arbitrator that grants permission to eat. Another approach is to use a timeout mechanism where a philosopher puts down a chopstick if they cannot acquire the second one within a specific time.

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

Ans: Starvation occurs when a philosopher is perpetually unable to acquire both chopsticks due to other philosophers continuously using them. This can happen if the resource allocation is unfair or if certain philosophers have higher priority access. To prevent starvation, techniques like fair scheduling, priority-based execution, and

allowing each philosopher a turn to eat can be used. By ensuring that every philosopher gets a chance to eat in a finite amount of time, starvation can be effectively mitigated.

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

Ans: Semaphores are used to control access to the shared chopsticks, ensuring proper synchronization between philosophers. A semaphore can be associated with each chopstick, initialized to 1, indicating availability. Before picking up a chopstick, a philosopher performs a "wait" operation, and after finishing eating, they perform a "signal" operation to release the chopsticks. This prevents multiple philosophers from using the same chopstick simultaneously and helps avoid race conditions, ensuring smooth and coordinated execution.

❖ **References:**

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