Maharashtra State Board of
Technical Education
(MSBTE)
Govt. of Maharashtra

subject name : OSY

Course name : CO

Academic Year:- 2025-26

Semester: FIFTH

Sr. No	Roll No. (Sem-V)	Full Name of Students	Enrollment No.	Seat No. (Sem-v)
1	**48**	**Aryan Yashawant Chawhan**	**23112090383**	**125749**

Under the Guidance of

Ms.Shraddha Chaudhary

In

3 Years of Diploma in Engineering & Technology of Maharashtra State

Board of Technical Education, (Mumbai)

At

SHIVAJIRAO S. JONDHALE POLYTECHNIC, ASANGAON

Seal of Institute

Maharashtra State Board of
Technical Education
(MSBTE)
Govt. of Maharashtra

MAHARASHATRA STATE BOARD OF TECHNICAL EDUCATION, MUMBAI

CERTIFICATE

This is to the certify Mr.**Aryan Yashawant Chawhan.** Roll No. 48 of **fifth Semester** of **Computer Engineering Diploma Programme In Engineering & Technology** at **Shivajirao S. Jondhle Polytechnic Asangaon (EAST) Shahapur 421601** has completed the **Micro Project Satisfactorily in Subject OSY In the academic year 2023 prescribed curriculum. Of K Scheme.**

Place: Asangaon
Date : **/ / 2023**

Enrolment No: 23112090383
Exam seat No: 125749

Project Guide | Head of the Department | Principal

Seal of Institute

INDEX

- **Acknowledgment**

It gives me great pleasure to submit this project report on “(Dining Philosophers Problem Implementatio)”. I express my sincere thanks to my project guide Respected Ms.Shraddha Chaudhary.

Department of Computer Engineering, for the constant support and guidance given to us throughout the course of this report. She has been a constant source of inspiration for us. I would like to express honest gratitude to Respected Mrs. Ashwini Dhage, Head of the Department of Computer Engineering for their guidance and time for valuable suggestions and providing constant support throughout this work.

We also take this opportunity to acknowledge the contribution of all faculty members of the department for their assistance and cooperation during the development of our project. I am also very much thankful to Respected Principle Dr. Anwesh K. Vikrunwar who has been constant source of inspiration to complete the work.

Last but not the least, we acknowledge our family members for their encouragement in the completion of the work.

Name Of Student:-

Aryan Yashawant Chawhan.

• Abstract

The **Dining Philosophers Problem** is one of the most famous classical synchronization problems in operating systems. It illustrates the challenges of allocating limited shared resources among multiple processes without causing deadlock or starvation. In this project, five philosophers sit around a table, each alternating between two states—thinking and eating. However, each philosopher requires two forks (shared resources) to eat, leading to a need for proper synchronization.

The main objective of this project is to design and implement a system that allows philosophers to share resources efficiently while preventing common concurrency issues such as **deadlock**, **race conditions**, and **starvation**. The solution uses **semaphores** and **mutex locks** to control access to shared resources, ensuring that no two neighboring philosophers eat simultaneously.

Through this implementation, we gain a deeper understanding of **inter-process communication**, **process synchronization**, and **resource allocation** in multitasking environments. The project demonstrates how synchronization mechanisms in operating systems maintain stability and fairness among concurrent processes.

• Introduction

The **Dining Philosophers Problem** is a classic synchronization and concurrency problem in operating systems, originally proposed by **Edsger W. Dijkstra** in 1965. It is designed to illustrate how processes can compete for limited shared resources in a way that could lead to **deadlock** or **starvation** if not properly managed. The problem is set around a group of philosophers sitting at a round table, alternating between thinking and eating. Between each pair of philosophers lies one fork, and each philosopher needs two forks to eat. The main challenge is ensuring that all philosophers can eat without causing a situation where everyone is waiting indefinitely.

In operating systems, this problem serves as an analogy for **process synchronization** and **resource allocation**. Each philosopher represents a process, and each fork represents a shared resource such as memory, files, or I/O devices. The system must ensure that processes access shared resources in a controlled manner to avoid **race conditions**, **deadlocks**, or **resource starvation**. Thus, the Dining Philosophers Problem provides a foundation for understanding how operating systems manage concurrency and maintain system stability.

The implementation of this problem involves the use of **semaphores** and **mutex locks**, which are fundamental tools for synchronizing processes. These synchronization primitives ensure that only one process can access a shared resource at a time, thus preventing conflicts. By carefully managing the order and timing of resource allocation, we can design a system where all philosophers (or processes) get a fair chance to eat without leading to deadlock or indefinite waiting. This approach reflects real-world scheduling and synchronization strategies used by operating systems.

Through this project, students gain practical knowledge about process communication, synchronization, and concurrency control mechanisms. It helps in understanding how operating systems handle multiple processes running simultaneously while sharing limited system resources efficiently. The **Dining Philosophers Problem** not only enhances theoretical understanding but also improves problem-solving skills in managing concurrent systems, making it an essential part of studying operating system design and implementation.

Objectives of the Project

1. **To understand process synchronization:**
   The main objective of this project is to study and implement synchronization techniques that allow multiple processes to work together without interference or conflict.

2. **To simulate the Dining Philosophers Problem:**
   The project aims to represent a real-time synchronization problem where multiple entities (philosophers) share limited resources (forks), highlighting how operating systems manage resource sharing.

3. **To prevent deadlock and starvation:**
   One of the core goals is to design a system that avoids situations where processes wait indefinitely for resources, ensuring that every process gets an equal opportunity to execute.

4. **To implement semaphores and mutex locks:**
   The project focuses on using synchronization tools like semaphores and mutexes to control access to shared resources effectively.

5. **To enhance understanding of concurrent process execution:**
   The project provides practical experience with concurrent programming concepts, helping students understand how multiple processes operate simultaneously and coordinate efficiently.

6. **To demonstrate real-world applications of OS concepts:**
   This project helps bridge the gap between theoretical concepts and real-world implementation of operating system functions such as process scheduling, communication, and synchronization.

• Theory and Background

The **Dining Philosophers Problem** is one of the most well-known examples used to explain **process synchronization** in operating systems. It was proposed by **Edsger W. Dijkstra** in 1965 to illustrate the challenges of sharing limited resources among multiple processes in a concurrent system. The problem describes five philosophers sitting around a round table, alternating between two states—**thinking** and **eating**. Between each pair of philosophers is a single fork, and each philosopher requires two forks to eat. This simple scenario creates a complex situation where improper synchronization can lead to **deadlock**, **starvation**, or **race conditions**.

In the context of operating systems, each philosopher represents a **process**, while each fork represents a **shared resource**. The goal of the operating system is to allow each process to access the resources it needs without causing conflicts or indefinite waiting. If every process holds one resource while waiting for another, the system enters a **deadlock**—a situation where no process can proceed. Therefore, the Dining Philosophers Problem serves as a practical demonstration of how to design systems that manage shared resources safely.

To address this problem, various synchronization techniques such as **mutex locks**, **semaphores**, and **monitors** are used.

- A **mutex (mutual exclusion lock)** ensures that only one process can access a shared resource at a time, preventing simultaneous access.
- **Semaphores** act as signaling mechanisms that coordinate processes, helping to prevent both deadlock and starvation.
- In some implementations, solutions involve ordering resource acquisition or limiting the number of philosophers that can eat simultaneously to ensure fairness.

Understanding this problem is fundamental to mastering key operating system concepts such as **process synchronization**, **inter-process communication**, **resource allocation**, and **deadlock handling**. These concepts are critical in real-world systems like databases, networking, and multitasking environments, where multiple processes operate concurrently. The Dining Philosophers Problem thus provides both a theoretical and practical framework for learning how operating systems manage concurrency effectively.

• Implementation

The implementation of the **Dining Philosophers Problem** is carried out in the **C programming language** on a **Linux environment** using **threads** and **semaphores** for synchronization. The problem aims to simulate five philosophers sitting around a table, alternating between thinking and eating, while ensuring that no two neighboring philosophers eat at the same time and no deadlock occurs.

Steps Involved:

1. Include the required header files: stdio.h, pthread.h, and semaphore.h.
2. Define constants such as the number of philosophers (e.g., 5).
3. Create an array of semaphores to represent forks.
4. Create philosopher threads using pthread_create().
5. Each philosopher will:
   - Think for a while.
   - Wait (use semaphore wait) for the left and right forks.
   - Eat for a short time.
   - Release the forks (use semaphore post).
6. Use synchronization to ensure that no two neighboring philosophers eat simultaneously.
7. Use pthread_join() to wait for all threads to complete.

Program Code (C Language):

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>


#define NUM_PHILOSOPHERS 5


sem_t forks[NUM_PHILOSOPHERS];
```

```
pthread_t philosophers[NUM_PHILOSOPHERS];

void* philosopher(void* num) {
   int id = *(int*)num;
   while(1) {
      printf("Philosopher %d is thinking...\n", id);
      sleep(1);

      // Pick up left fork
      sem_wait(&forks[id]);
      // Pick up right fork
      sem_wait(&forks[(id + 1) % NUM_PHILOSOPHERS]);

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

      // Put down forks
      sem_post(&forks[id]);
      sem_post(&forks[(id + 1) % NUM_PHILOSOPHERS]);

      printf("Philosopher %d finished eating and is thinking again...\n", id);
   }
   return NULL;
}

int main() {
   int i, ids[NUM_PHILOSOPHERS];
```

```
    // Initialize semaphores
    for (i = 0; i < NUM_PHILOSOPHERS; i++)
        sem_init(&forks[i], 0, 1);

    // Create philosopher threads
    for (i = 0; i < NUM_PHILOSOPHERS; i++) {
        ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &ids[i]);
    }

    // Wait for threads (infinite loop, can be terminated manually)
    for (i = 0; i < NUM_PHILOSOPHERS; i++)
        pthread_join(philosophers[i], NULL);

    // Destroy semaphores (optional)
    for (i = 0; i < NUM_PHILOSOPHERS; i++)
        sem_destroy(&forks[i]);

    return 0;
}
```

- **Output:**

When the program is executed, the output appears as follows (sample run):

```
Philosopher 0 is thinking...

Philosopher 1 is thinking...

Philosopher 2 is thinking...

Philosopher 3 is thinking...

Philosopher 4 is thinking...

Philosopher 1 is eating...

Philosopher 3 is eating...

Philosopher 1 finished eating and is thinking again...

Philosopher 0 is eating...

Philosopher 4 is eating...

Philosopher 2 is eating...
```

This output confirms that the philosophers take turns thinking and eating, with proper synchronization. No two neighboring philosophers eat at the same time, and no **deadlock** occurs.

• Result

The **Dining Philosophers Problem Implementation** was successfully carried out using **C language**, **threads**, and **semaphores** in a **Linux environment**. The program demonstrated the proper functioning of process synchronization and resource sharing among concurrent processes.

- Each philosopher alternates between **thinking** and **eating**, ensuring that no two neighboring philosophers eat simultaneously.
- The use of **semaphores** effectively prevented **deadlock** and **starvation**, allowing all philosophers to get a fair chance to eat.
- The output of the program clearly shows philosophers picking up and releasing forks in a synchronized manner, with proper timing between thinking and eating cycles.
- The simulation reflects real-world scenarios where multiple processes compete for limited resources, demonstrating how operating systems manage concurrency efficiently.

- ## Applications of Dining Philosophers Problem

The **Dining Philosophers Problem** is a classical example of process synchronization and resource allocation, and its concepts have several real-world applications in computer systems and operating systems. Some key applications include:

1. **Operating System Resource Management:**
   The problem illustrates how an operating system manages **multiple processes competing for limited resources**, such as CPU, memory, or I/O devices, without causing deadlock or starvation.
2. **Concurrent Programming:**
   It helps developers design **thread-safe programs** where multiple threads share resources, ensuring proper synchronization and avoiding race conditions.
3. **Database Systems:**
   In databases, multiple transactions often require access to shared data. Techniques learned from this problem (mutexes, semaphores) are applied to **prevent deadlock** and ensure **transaction consistency**.
4. **Networking and Communication Systems:**
   In networked systems, multiple processes or servers may request shared communication channels or bandwidth. The concepts from the Dining Philosophers Problem help in **efficient scheduling and resource allocation**.
5. **Embedded Systems and Real-Time Systems:**
   Systems with limited resources, such as printers, sensors, or actuators, rely on **synchronization mechanisms** inspired by this problem to avoid conflicts and ensure smooth operation.
6. **Teaching Tool:**
   It is widely used in **academics** to demonstrate **process synchronization, deadlock, and concurrency problems** in a clear and intuitive way.

- ## Conclusion

The **Dining Philosophers Problem** is a classic example used to understand **process synchronization**, **deadlock prevention**, and **resource allocation** in operating systems. Through this project, we successfully simulated multiple philosophers sharing limited resources, demonstrating how **semaphores** and **mutex locks** can coordinate access to shared resources effectively.

The implementation shows that proper synchronization ensures that no two neighboring philosophers eat at the same time, preventing **deadlock** and **starvation**, while allowing all processes to operate fairly. This practical exercise helps students understand how operating systems manage **concurrent processes** and maintain system stability in real-world scenarios.

Overall, the project provides valuable insights into **inter-process communication**, **concurrent programming**, and **resource management**, making it a foundational concept for studying and designing operating systems. It reinforces the importance of careful planning and synchronization when multiple processes share resources in a multitasking environment.