SIMULATE DINING-PHILOSHPHER PROBLEM USING SEMAPHORES

Project Based Learning (PBL) Report
for the course

Operating System(20CS22003)

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

By

P. Raziya(22R11A05H1)
Sk. Sania(22R11A05H6)
G. Sreeja(23R15A0517)

Under the guidance of Dr. K. Krishna Jyothi

Department of Computer Science and Engineering

Accredited by NBA

Geethanjali College of Engineering and Technology

(UGC Autonomous)

(Affiliated to J.N.T.U.H, Approved by AICTE, New Delhi)

Cheeryal (V), Keesara (M), Medchal.Dist.-501 301.

TABLE OF CONTENTS

S.No.	Contents	Page No
1	Abstract	3
2	Acknowledgement	4
3	Purpose of study	5
4	Introduction	6
5	Problem Statement	7
6	System Configuration	13
7	System Architecture	14
8	Proposed Solution	16
9	Implementation	17
10	Screen Short	21
11	Application	23
12	Implementation	25
13	Screen Short	28
14	Conclusion	35
15	Bibliography	36

ABSTRACT

The dining philosopher's problem is a classical synchronization challenge in computer science, illustrating the complexities of allocating shared resources without causing deadlock or resource starvation. This report presents a novel approach to simulating the dining philosopher's problem using semaphores within the context of an election process. By mapping the philosophers to election candidates and the forks to critical voting resources, we explore how semaphore-based synchronization mechanisms can be employed to manage concurrent access effectively. The simulation aims to ensure fair resource allocation, prevent deadlocks, and maintain system liveness, reflecting the intricacies of real-world electoral processes. Through this simulation, we demonstrate how semaphore operations—wait (P) and signal (V)—can orchestrate complex interactions among multiple candidates, ensuring orderly voting and resource distribution. The results provide insights into the effectiveness of semaphore-based synchronization in managing concurrent processes and highlight potential improvements for real-world applications in distributed systems and multi-agent environments.

ACKNOWLEDGEMENT

We would like to acknowledge and give my warmest thanks to our faculty “Dr K Krishna Jyothi” mam who made this work possible. Their guidance and advice carried us through all the stages of writing our project. We would also like to thank our classmates for letting our defence be an enjoyable moment, and for your brilliant comments and suggestions, thanks to you. We would also like to give special thanks to our families as a whole for their continuous support and understanding when undertaking my research and writing my project and providing the required equipment. The project would not have been successful without their cooperation and inputs.

PURPOSE OF STUDY

The primary purpose of this study is to explore the application of semaphore-based synchronization techniques by simulating the dining philosopher’s problem within the context of an election process. By mapping philosophers to election candidates and forks to critical voting resources, this study aims to demonstrate how semaphores can effectively manage concurrent access to shared resources. The simulation seeks to prevent common issues such as deadlock and resource starvation, ensuring that all candidates can access voting resources fairly and efficiently. This approach provides a practical illustration of semaphore utilization in managing complex interactions and maintaining system liveness in a competitive environment. Additionally, this study aims to bridge theoretical concepts with real-world scenarios, enhancing the understanding of concurrency issues in distributed systems. By modelling an election process, the study highlights the relevance and applicability of synchronization mechanisms in practical settings. The insights gained are intended to inform the design and implementation of robust systems, offering a foundational framework for future research. Ultimately, the study emphasizes the practical implications of semaphore-based synchronization, showcasing its potential to enhance the efficiency and reliability of concurrent systems across various domains

INTRODUCTION

The dining philosopher's problem, a classic synchronization issue in computer science, exemplifies the challenges of resource allocation and process coordination in concurrent systems. Traditionally, this problem involves a set of philosophers who alternately think and eat, requiring access to shared forks. The core challenge is to design a protocol that allows the philosophers to share resources without causing deadlock or starvation. This problem serves as a powerful metaphor for various real-world scenarios where multiple entities vie for limited resources.

In this report, we extend the dining philosophers problem to simulate an election process, using semaphores as the synchronization mechanism. Here, the philosophers are analogous to election candidates, and the forks represent critical voting resources. This simulation aims to explore how semaphore-based synchronization can effectively manage concurrent access to shared resources in a competitive environment. By implementing semaphore operations such as wait (P) and signal (V), we strive to ensure fair resource allocation, prevent deadlocks, and maintain system liveness. This study not only provides a practical demonstration of semaphores but also highlights their relevance and applicability in real-world scenarios, particularly in distributed systems and multi-agent environments.

PROBLEM STATEMENT

The dining philosopher’s problem is a classic example in computer science that deals with how multiple people (philosophers) can share a limited number of resources (forks) without running into issues like deadlock or unfair resource distribution. This problem helps us understand how to manage resource sharing in systems where many processes run at the same time. Can see the problem from figure 2 ,3,4,5. How it runs as infinity loops.

In this study, from fig1: we adapt the dining philosopher’s problem to simulate an election process. Here, the philosophers represent election candidates, and the forks are the critical voting resources they need. The main challenge is to create a system using semaphores (a tool for managing resource access) that ensures each candidate gets a fair chance to access the voting resources without causing deadlock or any candidate being unfairly blocked from accessing resources. This adapted problem aims to find practical solutions for managing shared resources in real-world scenarios, ensuring smooth and fair operations.

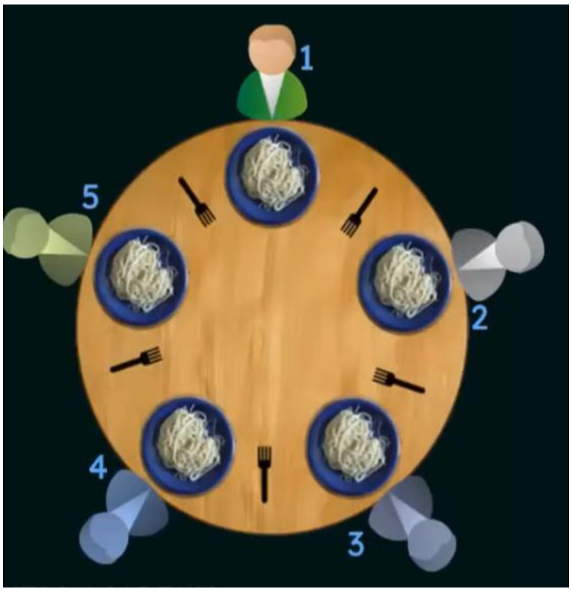

Fig1: the seating arrangement of philosophers

Problem Code:

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

sem_t forks [NUM_PHILOSOPHERS];
sem_t room; // Semaphore to limit the number of philosophers accessing forks at the same time

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

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

        sem_wait(&room); // Try to enter the room

        // Pick up left fork
        sem_wait(&forks[id]);
        printf ("Philosopher %d picked up fork %d (left)\n", id, id);
        // Pick up right fork
        sem_wait (&forks [(id + 1) % NUM_PHILOSOPHERS]);
        printf ("Philosopher %d picked up fork %d (right)\n", id, (id + 1) % NUM_PHILOSOPHERS);
```

```
    // Eating

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

        // Put down right fork
        sem_post (&forks [(id + 1) % NUM_PHILOSOPHERS]);
        printf ("Philosopher %d put down fork %d (right)\n", id, (id + 1) % NUM_PHILOSOPHERS);

        // Put down left fork
        sem_post(&forks[id]);
        printf("Philosopher %d put down fork %d (left)\n", id, id);

        sem_post(&room); // Leave the room

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

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];

  int ids[NUM_PHILOSOPHERS];
    // Initialize semaphores
```

```
  sem_init(&room, 0, NUM_PHILOSOPHERS - 1); // Allow up to NUM_PHILOSOPHERS-
1 philosophers to enter room
    int i;
        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 philosopher threads to finish (they never will in this example)
    for ( i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Destroy semaphores
    for ( i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_destroy(&forks[i]);
    }
    sem_destroy(&room);
    return 0;
}
```

SCREENSHORT

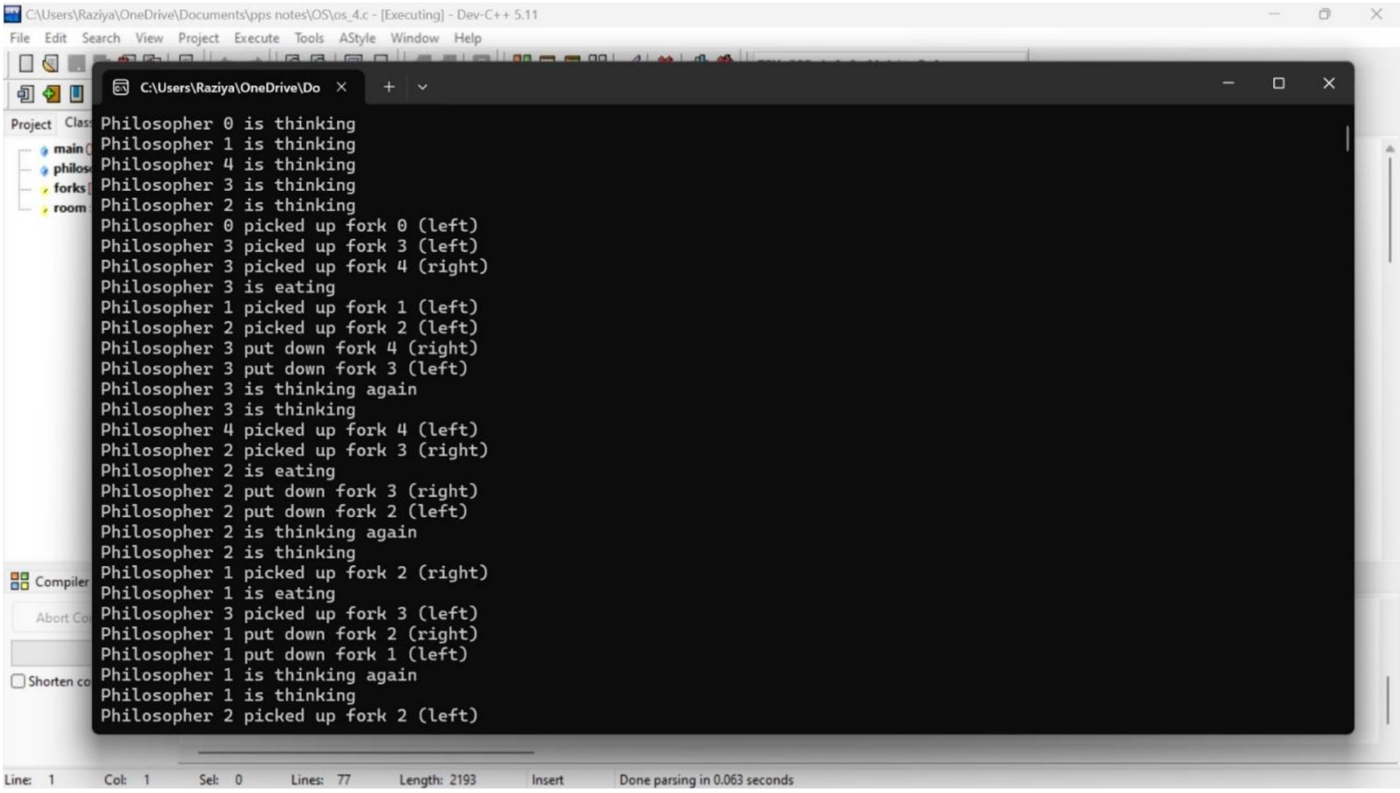

Fig2: this is the out of the problem which runs in infinity loop

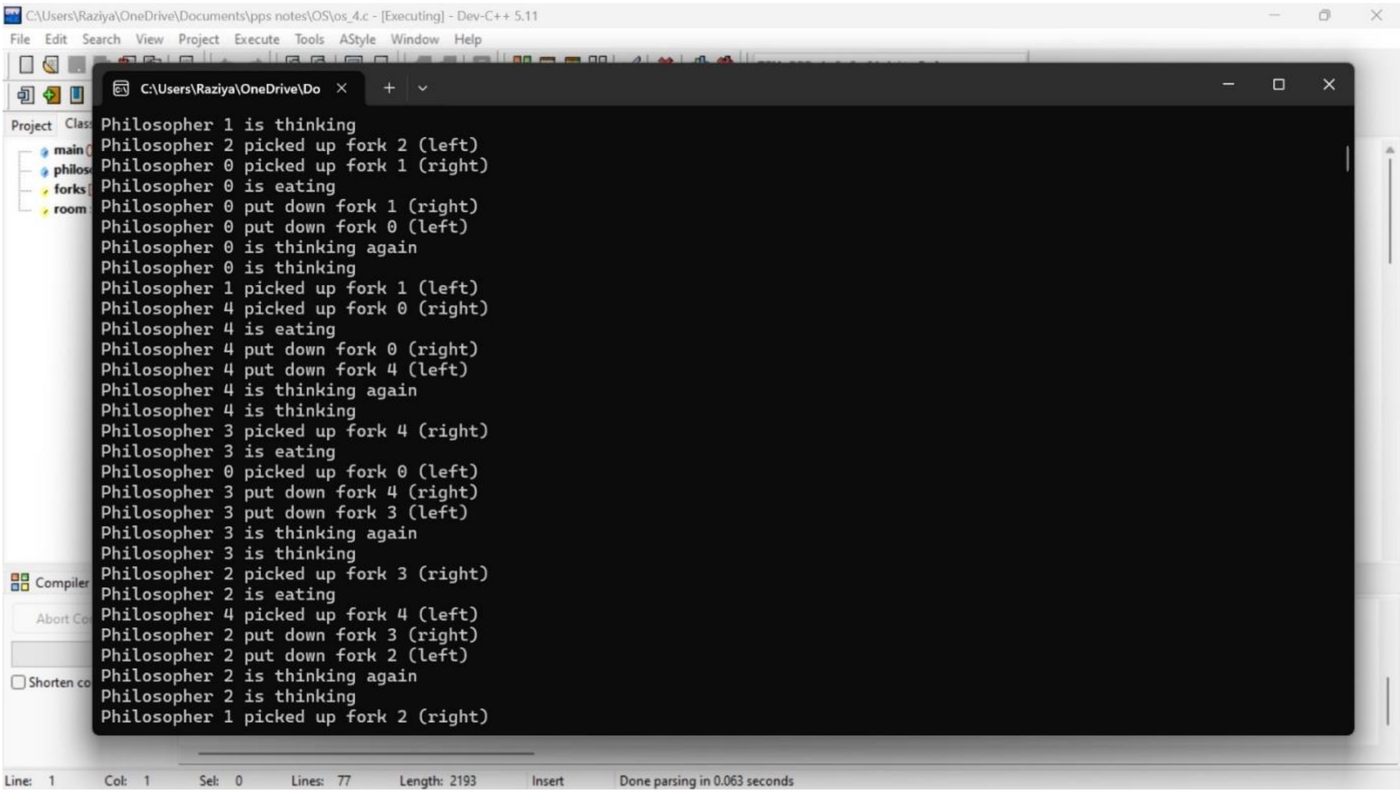

Fig3: this is the continues repetition after the single turn.

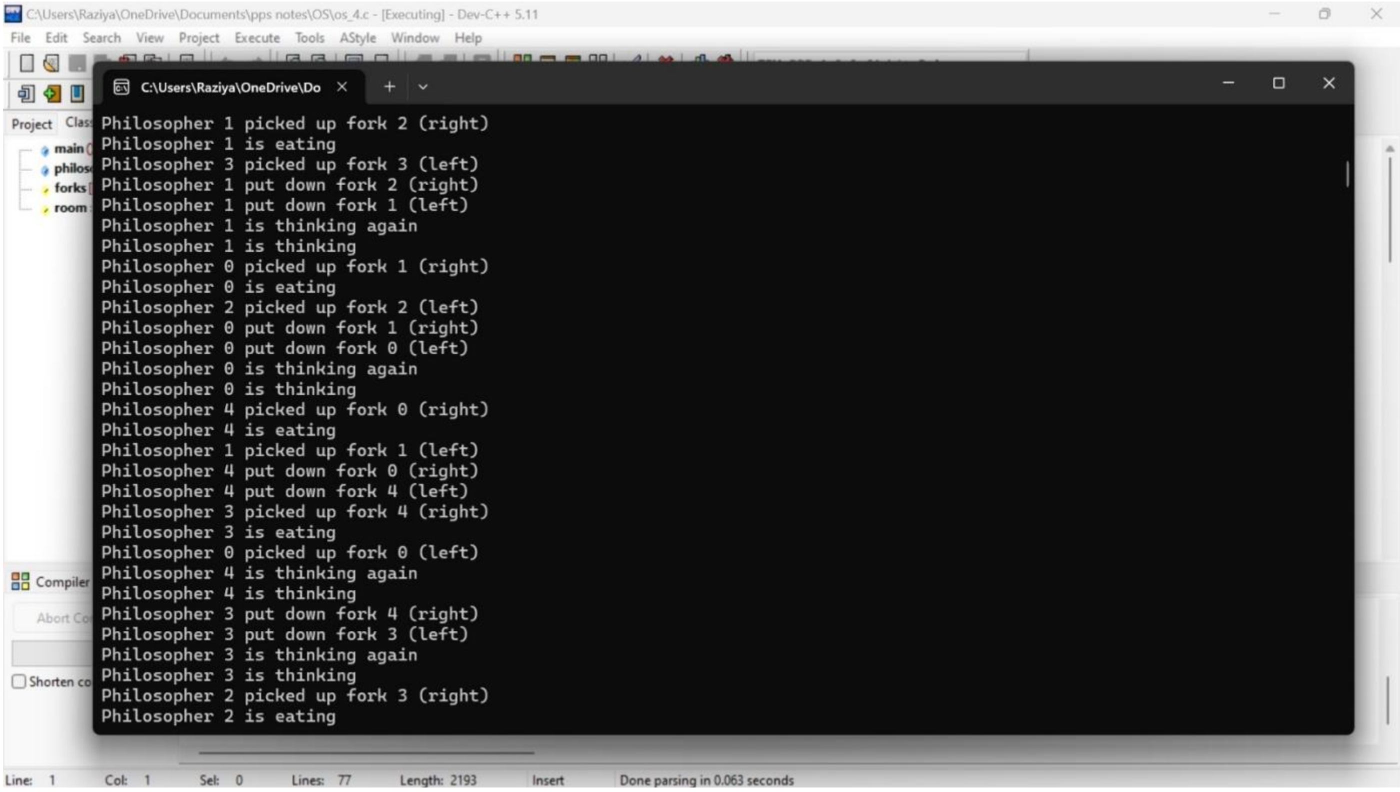

Fig4: infinity loop running after 15th notation

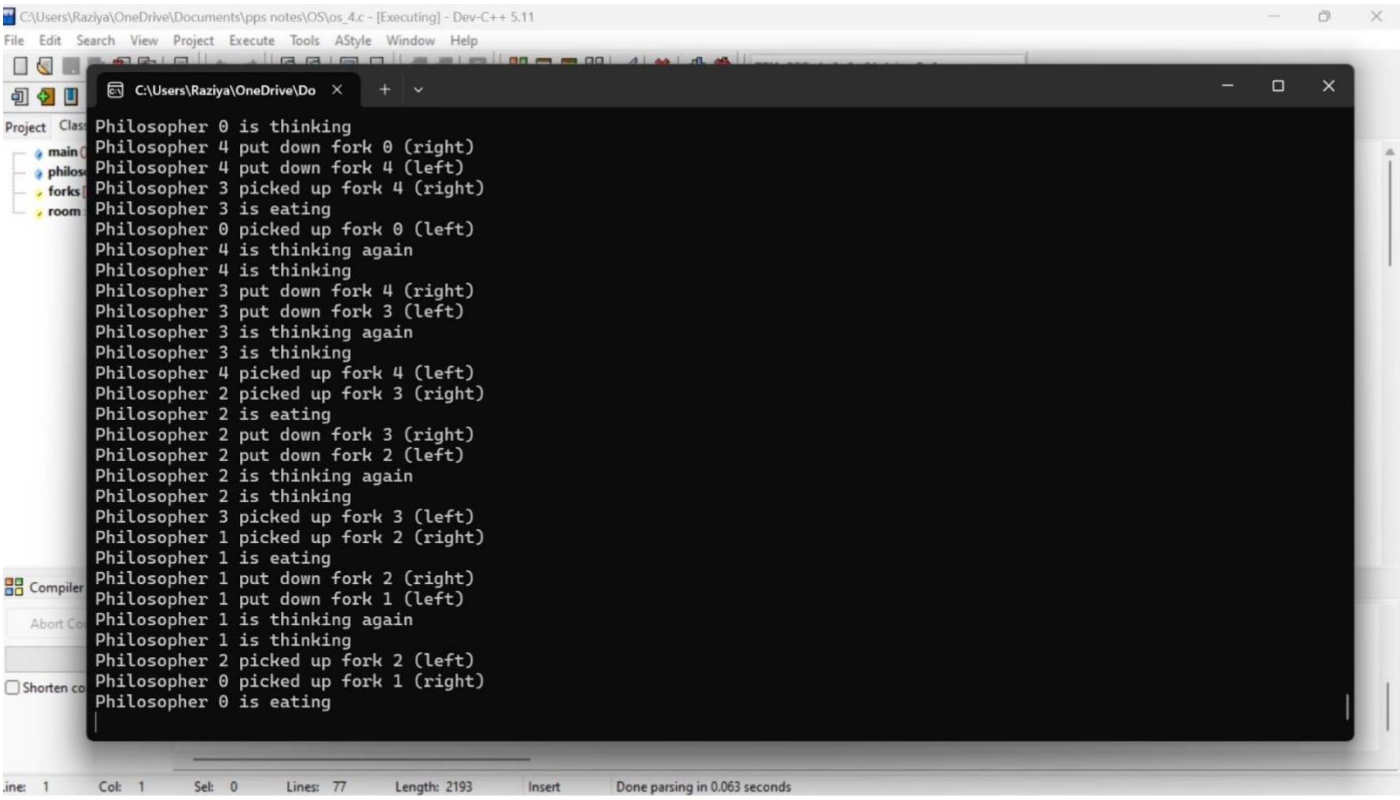

Fig5: the output does not end

SYSTEM CONFIGURATION

Hardware System Configuration

Component	Specification
Processor	Core i3 1005G1
Speed	4.40 GHz
RAM	4 GB
Hard Disk	1 TB (5400 rpm 2.5" SATA Hard Drive)
Keyboard	Dell Inspiring Keyboard
Mouse	Two or three button mouse

Software System Configuration

Component	Specification
Operating System	Windows 10 Home
Application Server	Tomcat 5.0/6.x
Front End	C Programming
Scripts	C-Script
Server-Side Script	C

SYSTEM ARTCHITECTURE

ASUMING A SITUATION:

Imagine a small town with five candidates running for office. Each candidate needs to gather signatures from five different voting districts to qualify for the election. There are only five signature collectors (one for each district), and each candidate can only work with one collector at a time.

In the traditional dining philosopher's problem, the candidates (philosophers) would need to pick up two forks to eat, but in our adapted problem, they need to work with one signature collector at a time. If two candidates try to get a signature from the same collector at the same time, neither can proceed, leading to a deadlock. We can see the flow of System by Fig6.

To solve this, we use semaphores. Each signature collector is protected by a semaphore that ensures only one candidate can access the collector at a time. When a candidate wants to get a signature, they perform a "wait" operation on the semaphore. If the semaphore is available, the candidate proceeds. If not, the candidate waits until the collector is free. After getting the signature, the candidate performs a "signal" operation to release the semaphore, making the collector available for the next candidate.

This way, we ensure that all candidates get their turn to gather signatures without any two candidates blocking each other, preventing deadlock and ensuring fair access to the voting resources.

FLOW OF THE SYSTEM

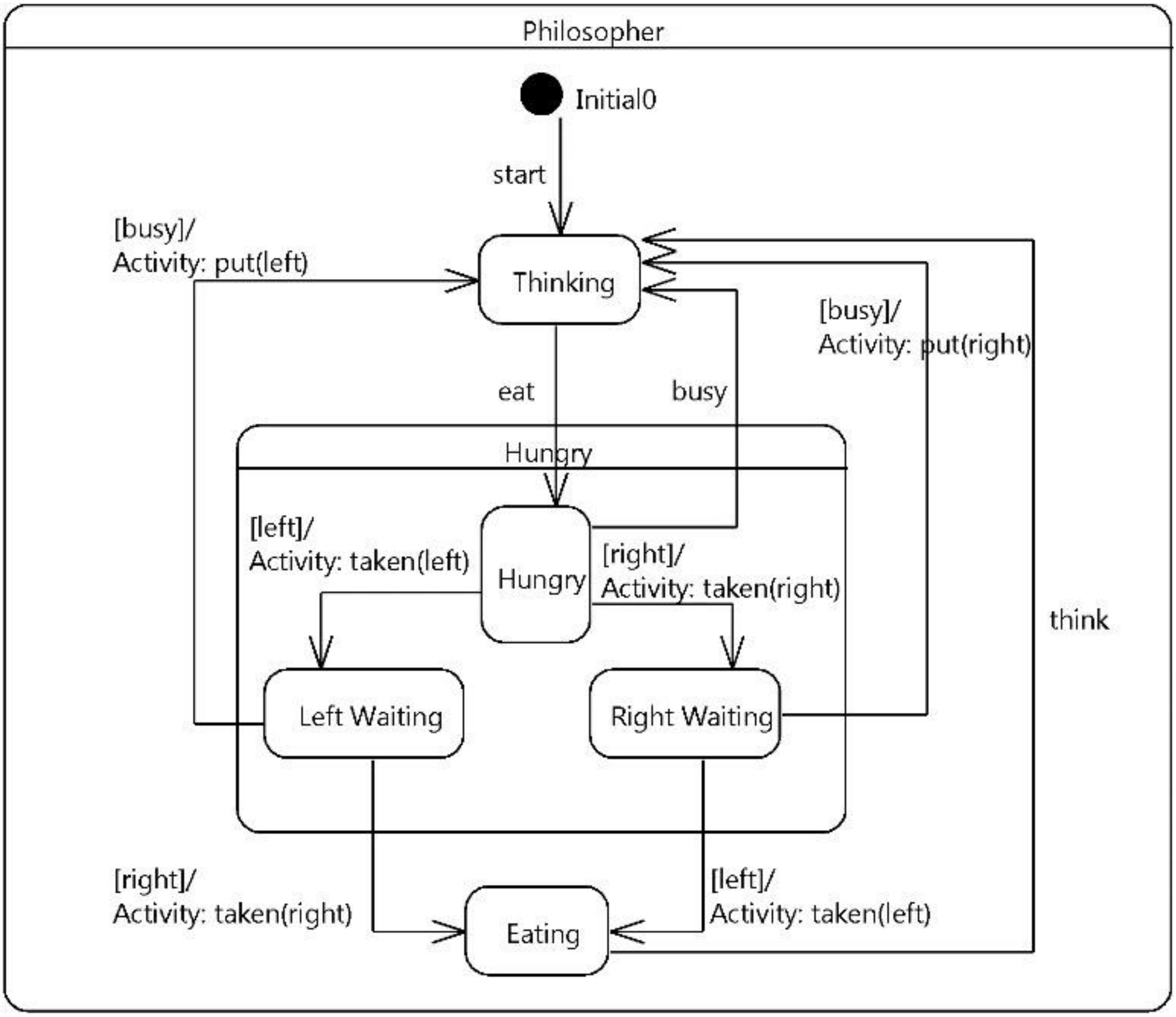

Fig6: this is the flow of our proposed solution and can see the processes how each fellow philosopher is Limited by 3.

PROPOSED SOLUTION

The code begins by defining constants for the number of philosophers (N) and their states (THINKING, HUNGRY, EATING). It also defines arrays to store the state of each philosopher, their IDs (ph. il), and the number of times each philosopher has eaten (eat_count).By fig7,8,9 we can assume the structure and output of the philosopher's. Semaphores mutex and S[N] are declared. mutex is used to ensure mutual exclusion when accessing shared resources, and S[N] is used to synchronize the philosophers.

test () function: This function checks if a philosopher can start eating. If a philosopher is hungry and its neighbors are not eating, it changes its state to eating and signals that it can start eating by posting to its semaphore.

take_fork() function: This function is called when a philosopher wants to eat. It sets the philosopher's state to hungry and then tries to eat by calling test(). After that, it waits until it can start eating by waiting on its semaphore.

put_fork () function: This function is called when a philosopher finish eating. It sets the philosopher's state to thinking, increments it eat count, and then calls test () for its neighbours.

philosopher () function: This function represents the behaviour of a philosopher. It keeps running in a loop until the philosopher has eaten EAT_LIMIT times. In each iteration, it alternates between thinking, taking forks, eating, and putting forks down.

main () function: It initializes semaphores, creates threads for each philosopher, and then waits for all threads to finish.

IMPLEMENTATION

Solution Code:

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

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N
#define EAT_LIMIT 3  // Each philosopher eats 3 times

int state[N];
int phil[N] = {0, 1, 2, 3, 4};
int eat_count[N] = {0}; // Counter for the number of times each philosopher has eaten

sem_t mutex;
sem_t S[N];

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

```
    state[phnum] = EATING;
   sleep(2);
    printf("Philosopher %d takes fork %d and %d\n", phnum + 1, LEFT + 1, phnum + 1);
     printf("Philosopher %d is Eating\n", phnum + 1);
     sem_post(&S[phnum]);
    }
}
void take_fork(int phnum)
{
    sem_wait(&mutex);
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);
    test(phnum);
    sem_post(&mutex);
    sem_wait(&S[phnum]);
    sleep(1);
}

void put_fork(int phnum)
{
    sem_wait(&mutex);
    state[phnum] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n", phnum + 1, LEFT + 1, phnum + 1);
printf("Philosopher %d is thinking\n", phnum + 1);
```

```
eat_count[phnum]++;
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}

void *philosopher(void *num)
{
    int *i = num;
    while(eat_count[*i] < EAT_LIMIT)
    {
       sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
    return NULL;
}
int main()
{
    int i;
    pthread_t thread_id[N];
    sem_init(&mutex, 0, 1);
    for(i = 0; i < N; i++)
        sem_init(&S[i], 0, 0);
```

```
 for(i = 0; i < N; i++)
   {
      pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
      printf("Philosopher %d is thinking\n", i + 1);
   }
   for(i = 0; i < N; i++)
{
   pthread_join(thread_id[i], NULL);
   printf("All philosophers have finished eating.\n");
   return 0;
}
```

SCREENSHORT

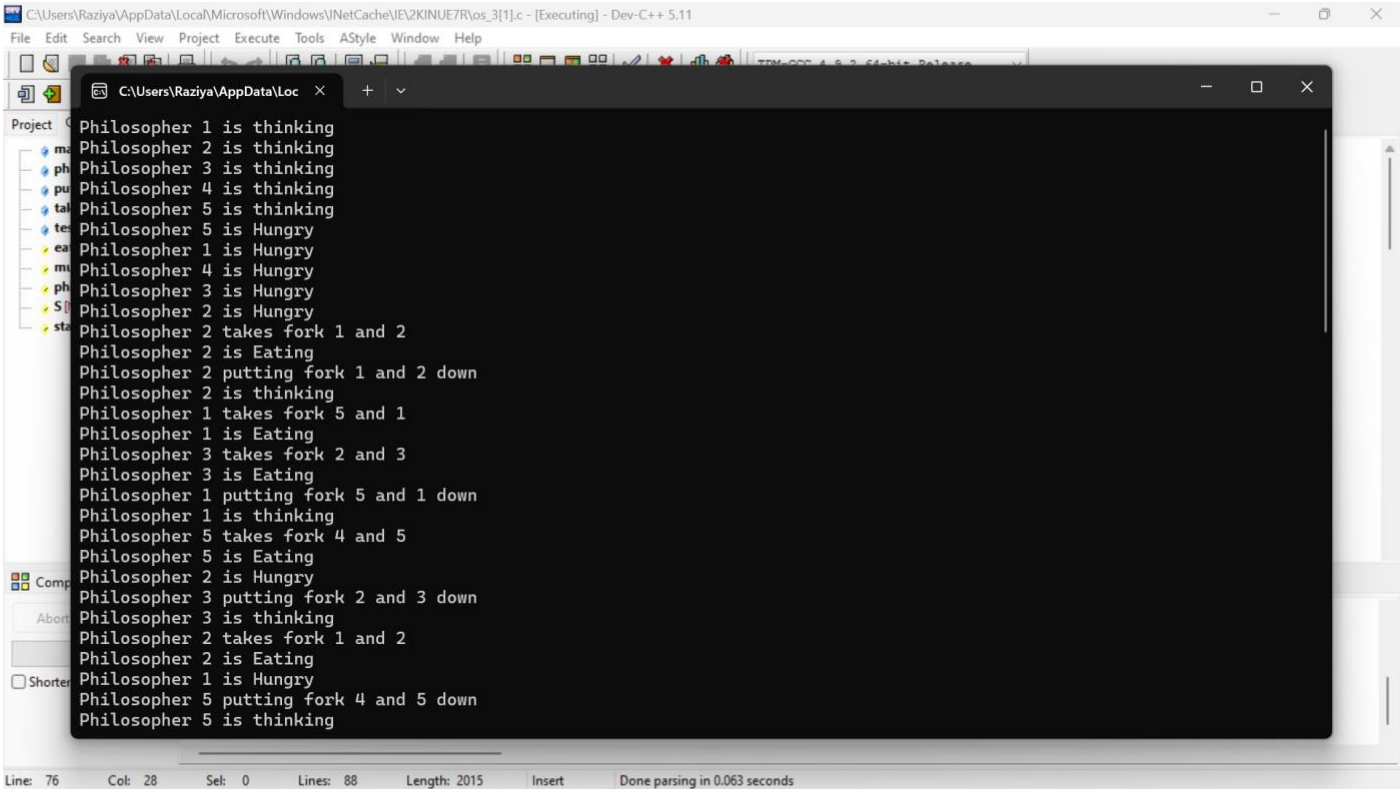

Fig7: The proposed solution in which each philosopher eats 3 times consider as per day.

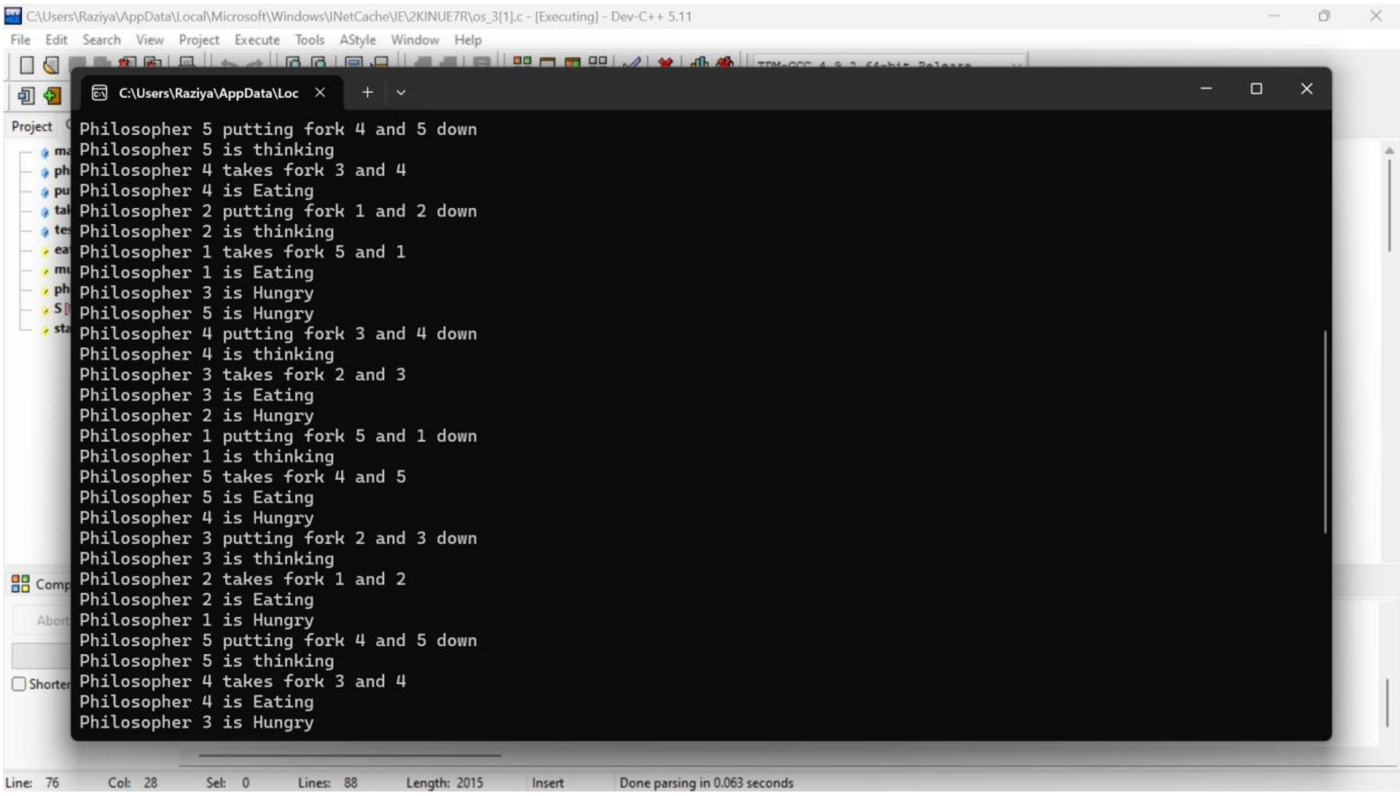

Fig 8: In this each philosopher Starts eating 2nd time

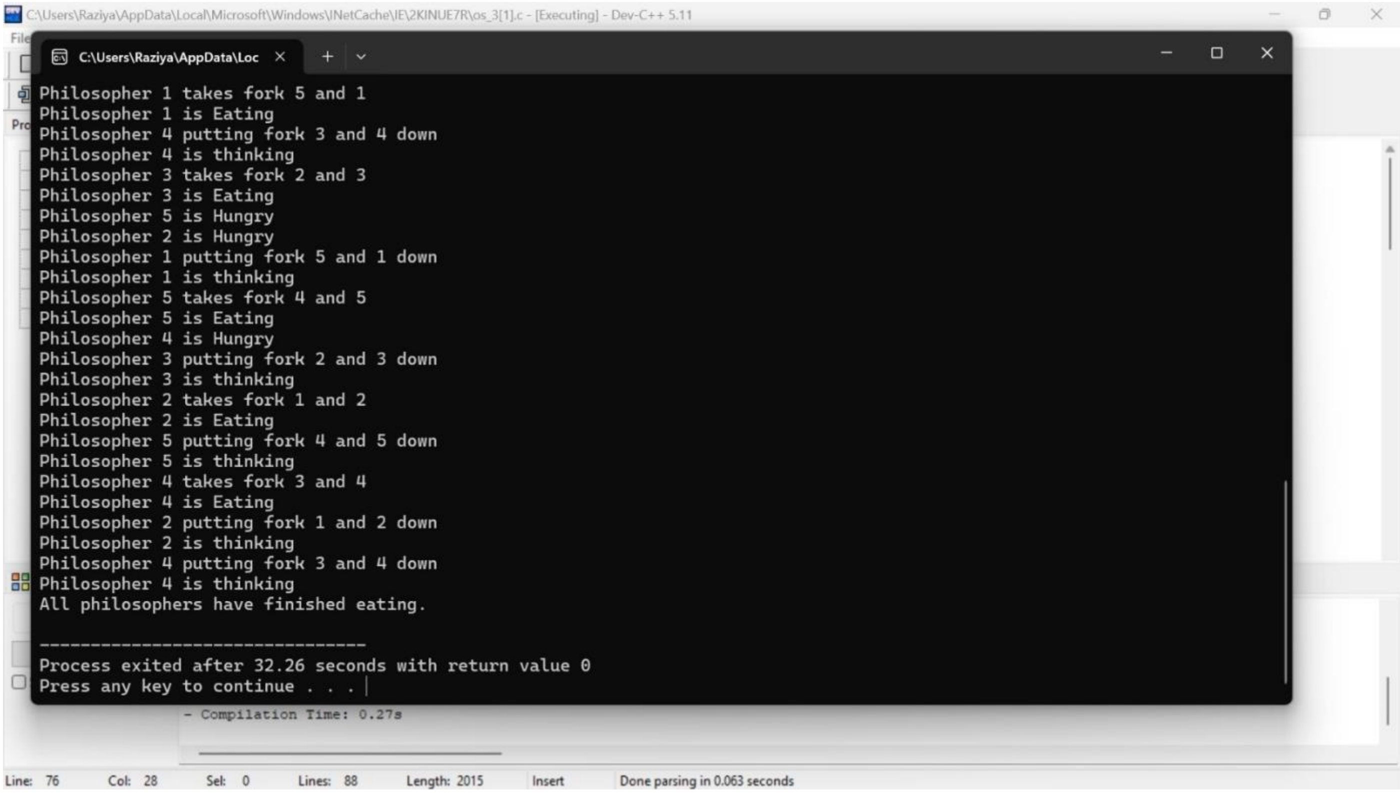

Fig9: In This the Philosopher starts eating 3rd i.e. is last time

APPLICATIONS:

The Dining Philosophers problem illustrates resource sharing and deadlock prevention in concurrent systems. Its principles are applied in various real-world scenarios:

Operating Systems: Manages resource allocation and concurrency control using semaphores and mutexes to prevent deadlocks and ensure fair access to resources like files and memory.

Database Systems: Handles concurrent transactions with locking mechanisms to maintain data consistency, using strategies to detect and prevent deadlocks.

Networking and Communication Protocols: Allocates bandwidth and manages resource reservations efficiently to avoid network contention and ensure smooth data transmission.

Manufacturing Systems: Coordinates tool sharing and production line synchronization to prevent resource contention and ensure smooth operation of automated systems.

Synchronizes threads to avoid race conditions and manage shared resources effectively, ensuring thread safety and optimal resource usage.

Real-Time Systems: Schedules tasks and handles priority inversion to meet deadlines and prevent conflicts over shared resources, ensuring timely task execution.

Election Algorithms: In distributed systems, election algorithms ensure that a single process (like a leader or coordinator) is selected from a group of processes to avoid conflicts and manage resources efficiently. Techniques from the Dining Philosophers problem help in designing these algorithms to avoid deadlocks and ensure fair election processes.

These applications use semaphore-based synchronization techniques to achieve efficient and fair resource allocation, preventing deadlocks and ensuring reliable operation. From fig10,11 you can consider people voting as 1^{st} type semaphores.

A semaphore typically consists of a non-negative integer counter and two atomic operations: wait (also known as P or down) and signal (also known as V or up). The wait operation decrements the semaphore's counter and blocks the calling thread if the counter becomes negative, while the signal operation increments the counter and unblocks a waiting thread if necessary.

In conclusion, from figure 12,13,14 we can see the end of voting and consider the all semaphores done and results are announced. The Dining Philosophers problem effectively demonstrates the complexities of managing concurrent resource allocation and the potential pitfalls of deadlock and resource contention in a shared environment. By utilizing semaphores to control access to shared resources (forks), we can ensure mutual exclusion and prevent deadlock, thereby allowing multiple processes (philosophers) to operate concurrently without conflict. This approach not only illustrates key concepts in concurrent programming but also provides a practical framework for addressing similar synchronization challenges in real-world applications.

IMPLEMENTATION

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM_CANDIDATES 5
sem_t booths[NUM_CANDIDATES];
sem_t room; // Semaphore to limit the number of candidates accessing booths at the same time
void* candidate(void* num) {
    int id = *(int*)num;
    while (1) {
        printf("Candidate %d is preparing to vote\n", id);
        sleep(1);
        sem_wait(&room); // Try to enter the room
        // Pick up left booth
        sem_wait(&booths[id]);
        printf("Candidate %d picked up booth %d (left)\n", id, id);
        // Pick up right booth
        sem_wait(&booths[(id + 1) % NUM_CANDIDATES]);
        printf("Candidate %d picked up booth %d (right)\n", id, (id + 1) %
NUM_CANDIDATES);
       // Voting
        printf("Candidate %d is voting\n", id);
        sleep(1);
        // Put down right booth
        sem_post(&booths[(id + 1) % NUM_CANDIDATES]);
```

```c
  printf("Candidate %d put down booth %d (right)\n", id, (id + 1) % NUM_CANDIDATES);

        // Put down left booth
        sem_post(&booths[id]);
        printf("Candidate %d put down booth %d (left)\n", id, id);

        sem_post(&room); // Leave the room

        printf("Candidate %d is done voting and preparing again\n", id);
    }
    return NULL;
}

int main() {
    pthread_t candidates[NUM_CANDIDATES];
    int ids[NUM_CANDIDATES];
   int i;
    // Initialize semaphores
    sem_init(&room, 0, NUM_CANDIDATES - 1); // Allow up to NUM_CANDIDATES-1
candidates to enter room
    for (i = 0; i < NUM_CANDIDATES; i++) {
        sem_init(&booths[i], 0, 1);
    }
```

```
// Create candidate threads
    for (i = 0; i < NUM_CANDIDATES; i++) {
        ids[i] = i;
  pthread_create(&candidates[i], NULL, candidate, &ids[i]);
    }

    // Wait for candidate threads to finish (they never will in this example)
    for (i = 0; i < NUM_CANDIDATES; i++) {
        pthread_join(candidates[i], NULL);
    }

    // Destroy semaphores
    for (i = 0; i < NUM_CANDIDATES; i++) {
        sem_destroy(&booths[i]);
    }
    sem_destroy(&room);

    return 0;
}
```

SCREENSHORT

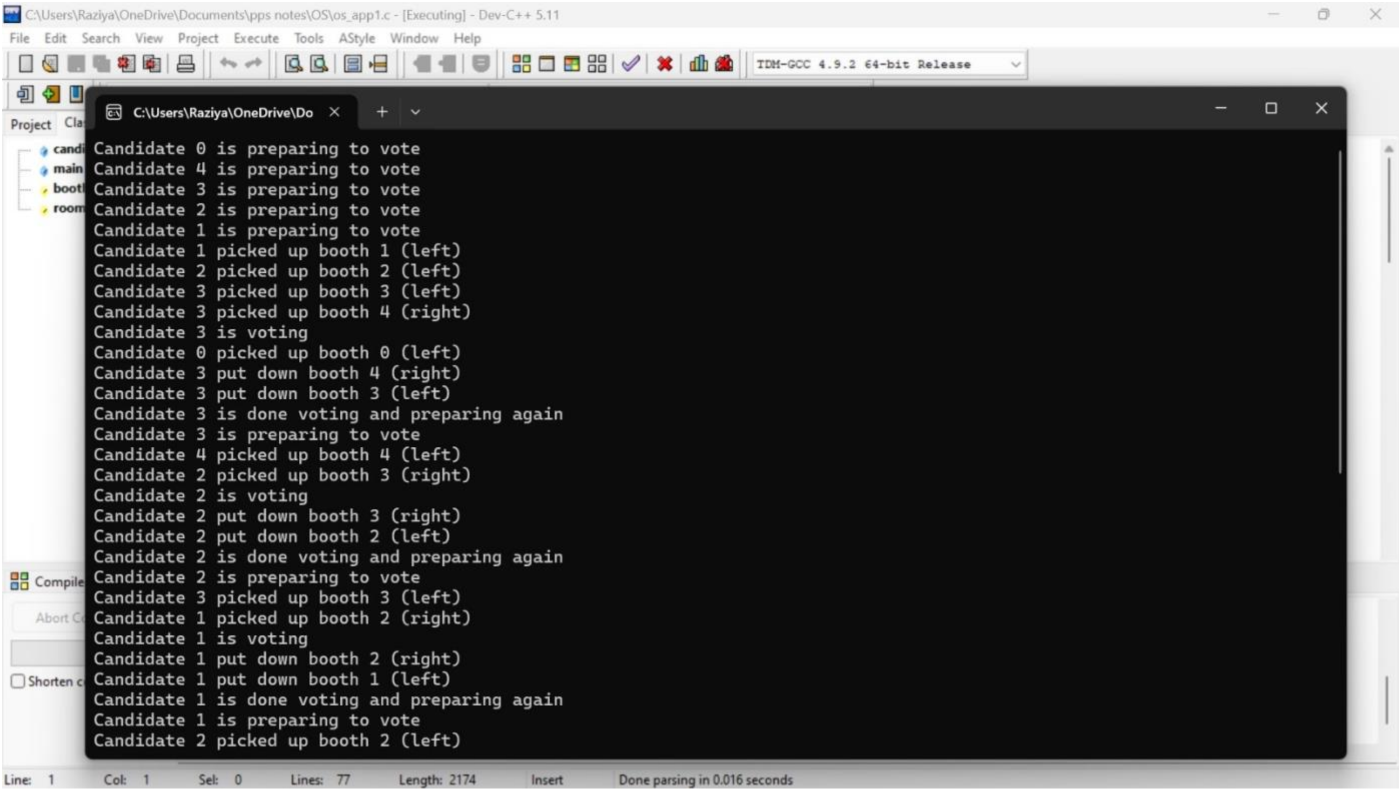

Fig10: In this 1st 3 areas the voters started their voting & leaving the booth.

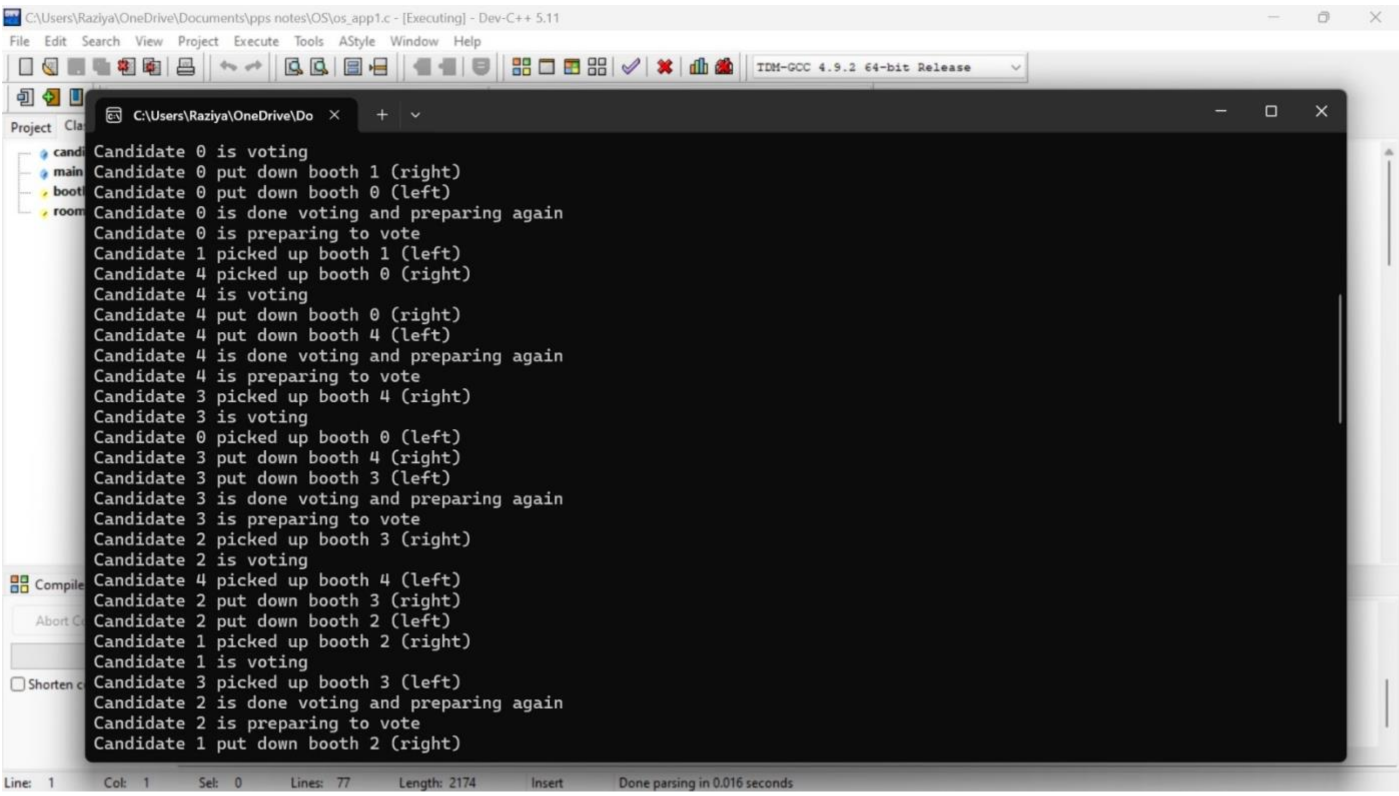

Fig11: In this next 4 booth voters are voting in the booth came (ri8) and left

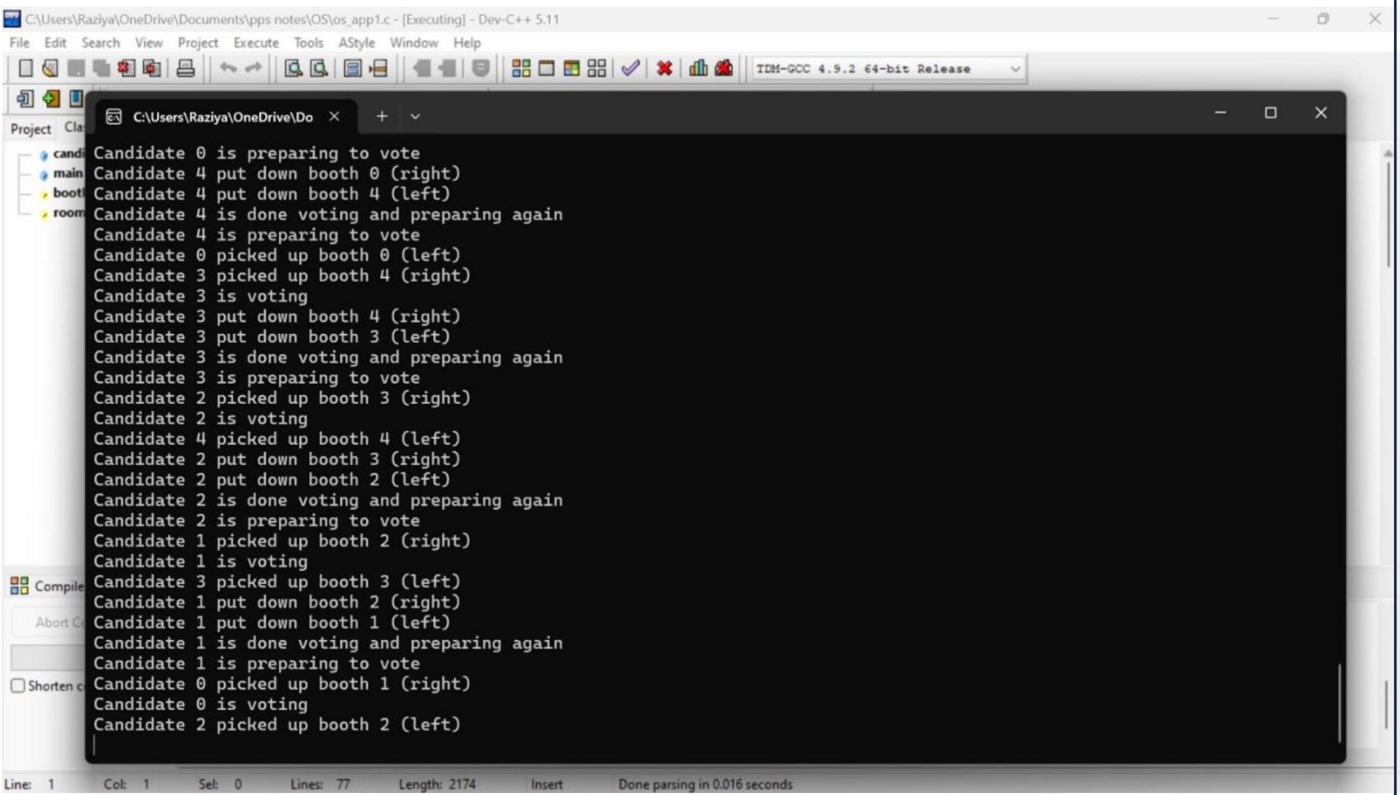

Fig12: In this the all the candidates of all the areas came and voted in there Respective booth and voting are completed.

```
// Continues………………
#define NUM_POLITICIANS 5
#define NUM_AREAS 10

sem_t booths[NUM_AREAS]; // Semaphores for voting booths in different areas
sem_t room; // Semaphore to limit the number of candidates voting at the same time

int votes[NUM_POLITICIANS] = {0}; // Votes received by each candidate
pthread_mutex_t vote_lock; // Mutex to protect vote counting

void* politician(void* num) {
    int id = *(int*)num;

    printf("Politician %d is preparing to vote\n", id);
    sleep(1);

    sem_wait(&room); // Try to enter the voting room

    int area = rand() % NUM_AREAS; // Randomly choose an area for the politician to vote
    sem_wait(&booths[area]);


    printf("Politician %d entered booth in area %d\n", id, area);
    // Voting
    printf("Politician %d is voting in area %d\n", id, area);
    sleep(1);
```

```
// Lock the votes array and update the vote count
    pthread_mutex_lock(&vote_lock);
    votes[id]++; // Increment the vote count for this politician
    pthread_mutex_unlock(&vote_lock);

    // Leave the booth
    sem_post(&booths[area]);
    printf("Politician %d left booth in area %d\n", id, area);

    sem_post(&room); // Leave the voting room

    printf("Politician %d is done voting and preparing again\n", id);

    // Exit after casting a vote for simplicity in this example
    return NULL;
}

int main() {
    pthread_t politicians[NUM_POLITICIANS];
    int ids[NUM_POLITICIANS];
     int i;
    // Initialize semaphores and mutex

  sem_init(&room, 0, NUM_AREAS); // Allow up to NUM_AREAS politicians to vote
simultaneously
    }
```

```
for (i = 0; i < NUM_AREAS; i++) {
    sem_init(&booths[i], 0, 1);

pthread_mutex_init(&vote_lock, NULL);

// Create politician threads
for ( i = 0; i < NUM_POLITICIANS; i++) {
    ids[i] = i;
    pthread_create(&politicians[i], NULL, politician, &ids[i]);
}

// Wait for politician threads to finish
for ( i = 0; i < NUM_POLITICIANS; i++) {
    pthread_join(politicians[i], NULL);
}

// Assign unique votes to each politician for demonstration
for ( i = 0; i < NUM_POLITICIANS; i++) {
    votes[i] = rand() % 100 + 1; // Random votes between 1 and 100
}

// Determine the winner
int max_votes = 0;
int winner = -1;
for ( i = 0; i < NUM_POLITICIANS; i++) {
```

```
        if (votes[i] > max_votes) {
    max_votes = votes[i];
            winner = i;
        }
    }

    // Print results in tabular form
    printf("\nElection Results:\n");
    printf("+----------------+---------------+\n");
    printf("| Politician ID  | Votes Received|\n");
    printf("+----------------+---------------+\n");
    for ( i = 0; i < NUM_POLITICIANS; i++) {
        printf("|      %d        |       %d       |\n", i, votes[i]);
    }
    printf("+----------------+---------------+\n");
    printf("The winner is Politician %d with %d votes.\n", winner, max_votes);

    // Destroy semaphores and mutex
    for (i = 0; i < NUM_AREAS; i++) {
        sem_destroy(&booths[i]);
    }
    sem_destroy(&room);
    pthread_mutex_destroy(&vote_lock);
    return 0;
}
```

SCREENSHORT

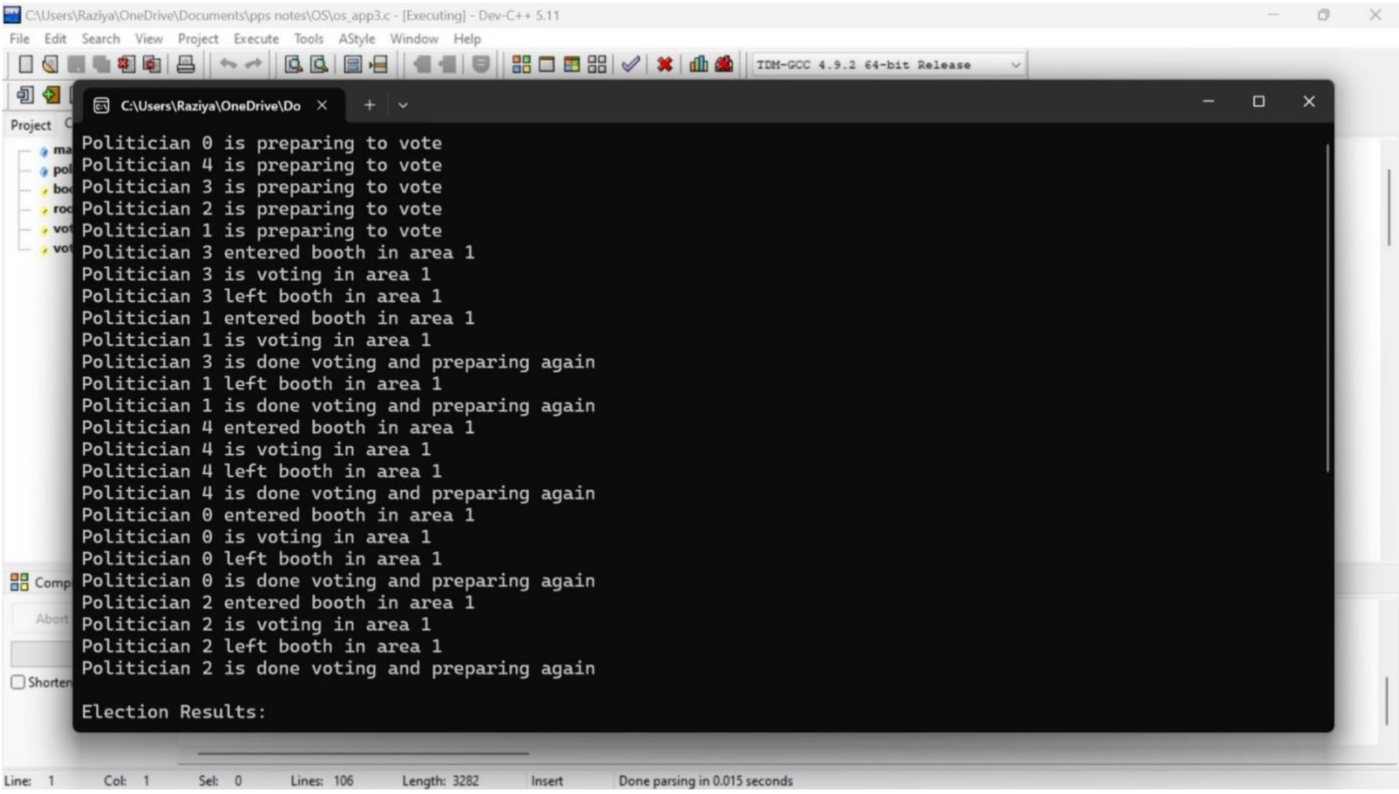

Fig13: Here the political Candidates are casting votes

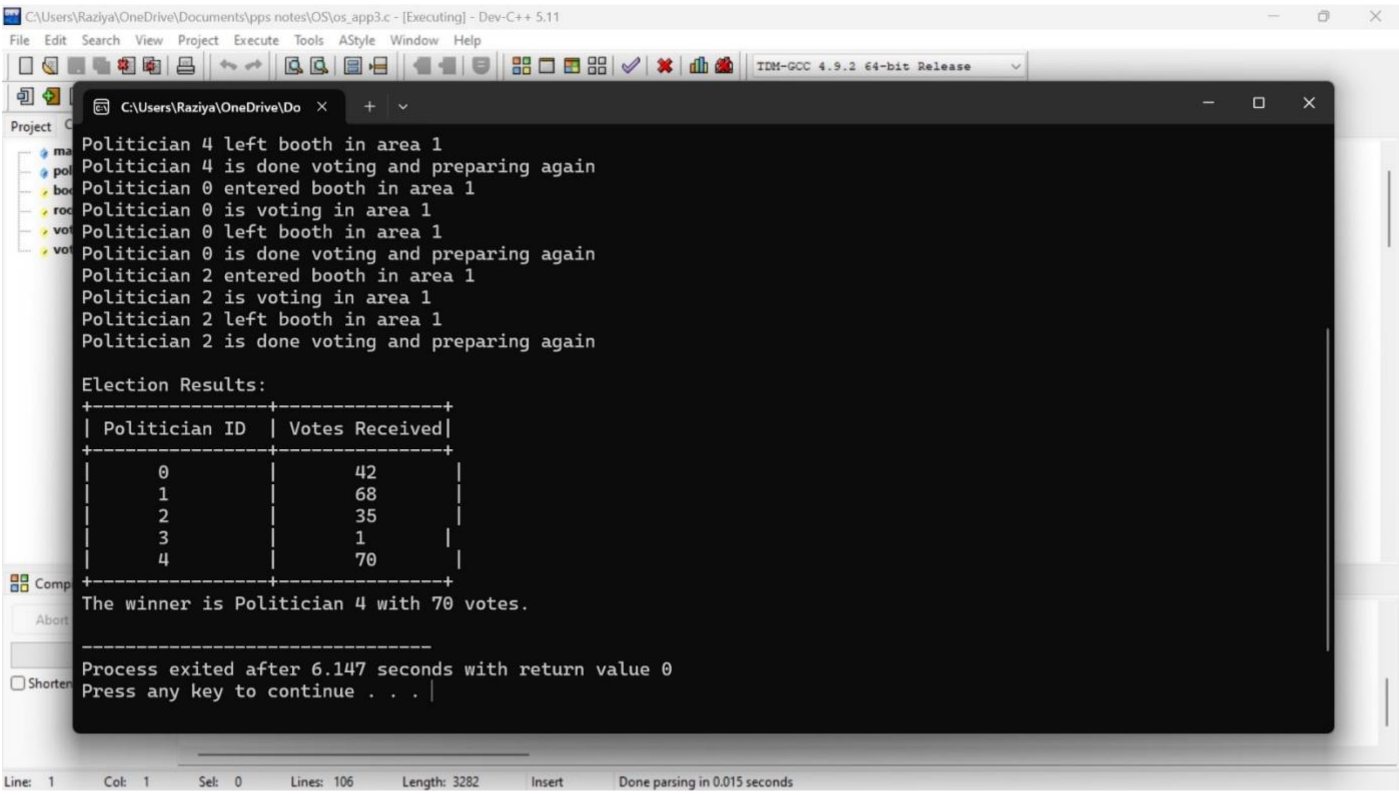

Fig14: Here we are declaring the results based on voters and candidates voting.

CONCLUSION

Hence, we concluded that the use of semaphores and mutexes for managing resource allocation and synchronization in concurrent systems, applied to voting processes. In the first example, candidates simulate the Dining Philosophers problem, using semaphores to ensure that only a limited number can enter the voting room simultaneously and manage booth access to avoid deadlock. The second example involves politicians voting in multiple areas, with semaphores controlling booth entry and a mutex ensuring accurate vote tallying without race conditions. These examples emphasize the importance of efficient resource management, preventing contention, and ensuring fair access. The practical implementation includes random vote assignments and tabular election results, demonstrating the techniques' relevance in real-world scenarios like operating systems, database management, and distributed systems. By employing synchronization mechanisms, the codes ensure orderly operation and reliable coordination of shared resources, critical for system stability and fairness.

BIBILOGRAHY

- www.google.com
- www.wikipedia.com
- www.gcet.library.com