

**Mini Project Report**

**of**

**Operating Systems Lab (CSE 3163)**

**Title: Deadlock Detection in Train Scheduling**

**SUBMITTED**

**BY**

**Devanshu Singh (Reg. No. 210905314)**

**Roll Number - 48**

**Gauri Saxena (Reg No. 210905346)**

**Roll Number - 55**

**Department of Computer Science and Engineering Manipal Institute of Technology,**

**Manipal, Karnataka 20th NOV 2023**



**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**Manipal**

**06/11/2023**

**CERTIFICATE**

This is to certify that the project titled **Deadlock Detection in Train Scheduling** is a record of the bonafide work done by **Devanshu Singh (210905314) and Gauri Saxena (210905346)** submitted in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology (B.Tech.) in COMPUTER SCIENCE & ENGINEERING of Manipal Institute of Technology, Manipal, Karnataka, (A Constituent Institute of Manipal Academy of Higher Education), during the academic year 2022- 2023.

**Name and Signature of Examiners:**

1. **Dr. Sucharitha Shetty, Assistant Professor - Selection Grade, CSE Dept.**

1. **Dr. P B Shanthi, Assistant Professor -Selection Grade, CSE Dept.**

**ACKNOWLEDGMENTS**

We extend our heartfelt appreciation to our mentors at MANIPAL INSTITUTE OF TECHNOLOGY, particularly **Dr. Sucharitha Shetty**, whose guidance and expertise have played a crucial role in the completion of this video streaming project. The foundation of knowledge in networking, socket programming, and multimedia processing that we utilized in this endeavour was cultivated through their insightful instruction. We are grateful for the support, encouragement, and invaluable insights provided by our educators, which significantly contributed to the success of our project. Their commitment to fostering a rich learning environment has been pivotal, and we acknowledge their impactful contributions with sincere gratitude.

**TABLE OF CONTENTS**

**CHAPTER 1: INTRODUCTION**

**CHAPTER 2: PROBLEM STATEMENT, DESCRIPTION AND THE OBJECTIVES**

**CHAPTER 3: ALGORITHM**

**CHAPTER 4: METHODOLOGY**

**CHAPTER 5: CODE**

**CHAPTER 6: FINAL OUTPUT**

**CHAPTER 7: LIMITATIONS**

**CHAPTER 8: CONCLUSION**

**CHAPTER 9: REFRENCES**

**INTRODUCTION**

The Train Scheduling with Deadlock Detection project is designed to simulate a railway junction where trains arrive, request tracks, pass through allocated tracks, and finally leave the junction. The project incorporates a robust scheduling mechanism using Banker's Algorithm to allocate tracks efficiently while considering the priority of each train. Additionally, it features a deadlock detection system to identify potential deadlock situations and a mechanism for resolution.

**Key Features:-**

1. **Resource Allocation with Banker's Algorithm:**
   1. The project employs the Banker's Algorithm to allocate tracks to arriving trains efficiently.
   2. Each train specifies its maximum required tracks, and the algorithm ensures that tracks are allocated in a way that avoids potential deadlock situations.
2. **Priority-Based Scheduling:**
   1. Trains are assigned priorities, and the scheduling algorithm considers these priorities when allocating tracks.
   2. Priority is a crucial factor in the decision-making process, allowing trains with higher priority to obtain tracks before others.
3. **Deadlock Detection and Resolution:**
   1. A dedicated thread periodically checks for potential deadlocks in the system.
   2. If a deadlock is detected, the system releases acquired locks to prevent a deadlock and potentially implements resolution strategies (placeholder logic is included).
4. **User Interaction:**
   1. Users can input the number of tracks and trains.
   2. Two options are provided for inputting maximum required tracks: manual input or random generation.
   3. Users are given flexibility in configuring the simulation parameters.
5. **Logging:**
   1. The project logs significant events, including train arrivals, track allocations, deallocations, and deadlock detection.
   2. Timestamped logs enhance traceability and facilitate debugging.
6. **Multithreading and Synchronization:**
   1. Utilizes multithreading with the pthread library to simulate concurrent train arrivals.
   2. Synchronization mechanisms, including mutex locks and semaphores, are implemented to ensure thread safety.
7. **Visualization:**
   1. Provides a visual representation of tracks and trains to enhance understanding.
   2. The display functions show the current status of tracks and the allocation of tracks to trains.

**Conclusion:**

The Train Scheduling with Deadlock Detection project offers a comprehensive simulation of a train scheduling system, combining efficient resource allocation, priority-based scheduling, and mechanisms to detect and potentially resolve deadlocks. This project provides insights into concurrent programming, synchronization, and resource management in a real-world scenario.

**Problem Statement, Description and Objectives:**

**Problem Statement:**

The project addresses the challenge of developing a robust train scheduling system for a railway junction, focusing on efficient resource allocation and deadlock detection. The primary objective is to create a simulation that mimics the arrival, track allocation, and departure of trains, ensuring a prioritized and deadlock-free scheduling process. The system must implement the Banker's Algorithm for resource allocation, consider the priority of trains in decision-making, and incorporate a mechanism to detect and potentially resolve deadlocks. Users should have the flexibility to input the number of tracks and trains, and the system should provide visual representation and logging for better understanding and debugging. The project aims to simulate a real-world scenario, offering insights into concurrent programming, synchronization, and resource management challenges.

**Description:**

The Train Scheduling with Deadlock Detection project offers a dynamic simulation of a railway junction, presenting a comprehensive solution to challenges in concurrent train scheduling. At its core is the robust implementation of the Banker's Algorithm, a key feature that optimizes resource allocation for arriving trains. The system introduces a unique priority-based scheduling mechanism, granting higher priority to certain trains for expedited track access. This prioritization not only enhances operational efficiency but also contributes to a streamlined and responsive scheduling process.

A pivotal aspect of the project is the incorporation of a deadlock detection system. A dedicated thread regularly scrutinizes the system's state, promptly identifying potential deadlocks. In case of detection, the system intelligently releases acquired locks to preemptively resolve the deadlock situation. This intricate mechanism adds a layer of sophistication to the simulation, ensuring the reliability and stability of the scheduling process.

User interaction is facilitated through flexible input options. Users can determine the number of tracks and trains involved, offering adaptability to different scenarios. Additionally, the project provides options for users to manually specify the maximum required tracks for each train or opt for the system to generate random values, promoting ease of use and customization.

To enhance the project's comprehensibility, detailed logging is implemented, timestamping critical events such as train arrivals, track allocations, and deadlock detection. Visualization features offer users an insightful representation of the dynamic allocation status of tracks to trains, fostering a deeper understanding of the simulation's inner workings.

In conclusion, the Train Scheduling with Deadlock Detection project serves as an advanced exploration into concurrent programming, synchronization, and resource management. By combining sophisticated algorithms with user-friendly interfaces and insightful visualizations, the project provides a rich learning experience in the complex realm of real-world train scheduling challenges.

**OBJECTIVES:**

* **Implement Banker's Algorithm:**
  + Develop a robust implementation of the Banker's Algorithm for efficient and safe resource allocation, considering the maximum required tracks for each arriving train.
* **Priority-Based Scheduling:**
  + Introduce a priority-based scheduling mechanism to ensure that higher-priority trains are given precedence in resource allocation, contributing to a responsive and prioritized scheduling system.
* **Deadlock Detection System:**
  + Implement a dedicated deadlock detection system that periodically checks the system's state to identify potential deadlocks and take preemptive measures to avoid deadlock situations.
* **User Interaction and Configuration:**
  + Provide a user-friendly interface that allows users to input the number of tracks and trains, offering flexibility in configuring the simulation for different scenarios.
* **Manual and Random Input Options:**
  + Implement options for users to manually input the maximum required tracks for each train or opt for the system to generate random values, enhancing user adaptability and ease of use.
* **Logging and Visualization:**
  + Develop a comprehensive logging system that timestamps significant events, providing users with detailed insights into train arrivals, track allocations, and deadlock detection. Implement visualizations for a clearer representation of the system's dynamics.
* **Multithreading and Synchronization:**
  + Utilize multithreading with the pthread library to simulate concurrent train arrivals. Implement synchronization mechanisms, including mutex locks and semaphores, to ensure thread safety during resource allocation and deallocation.

**ALGORITHM**

This algorithm provides an overview of the main steps and interactions within the train scheduling code.

* **Initialization:**
  + Initialize global variables, including arrays for tracks and trains, mutex locks, semaphore, and log file.
  + Open the log file for writing.
* **Setup Tracks:**
  + Call `setup\_tracks` function to initialize tracks, associated mutex locks, and the safety check lock.
* **User Input:**
  + Prompt the user to input the number of tracks and trains.
  + Provide options to choose between entering maximum required tracks manually or generating random values.
  + Validate user input for correctness.
* **Thread Creation:**
  + Create an array of train threads (`trains`) and an array of train IDs (`train\_ids`).
  + Use `pthread\_create` to create a thread for each train, executing the `train\_arrival` function.
* **Deadlock Detection Thread:**
  + Create a thread (`deadlock\_thread`) for deadlock detection, executing the `deadlock\_detection` function.
* **Display Initial Status:**
  + Display the initial status of tracks and trains using `display\_track\_status`, `display\_train\_status`, and `display\_tracks\_and\_trains` functions.
* **Train Arrival Simulation:**
  + Each train thread executes the `train\_arrival` function.
  + Simulate the train's approach, arrival, and attempt to acquire tracks using Banker's Algorithm.
  + If a potential deadlock is detected, release acquired locks and exit the thread.
  + If tracks are successfully allocated, simulate the train passing through the tracks.
  + Release the allocated tracks after passing through.
* **Deadlock Detection:**
  + The `deadlock\_detection` thread periodically checks for deadlocks.
  + If a deadlock is detected, print a message and potentially implement deadlock resolution logic (currently a placeholder).
* **Thread Joining:**
  + Wait for all train threads to finish using `pthread\_join`.
* **Display Final Status:**
  + Display the final status of tracks and trains using the same display functions as in step 6.
* **Cancel Deadlock Detection Thread:**
  + Cancel the `deadlock\_thread` using `pthread\_cancel`.
* **Lock Destruction:**
  + Destroy mutex locks for tracks and the safety check lock.
* **File Closure:**
  + Close the log file.
* **Program Exit:**
  + Exit the program.

**Notes:**

- The `train\_arrival` function simulates the behavior of each train, attempting to acquire tracks and releasing them after passing through.

- Deadlock detection is performed periodically in a separate thread (`deadlock\_thread`).

- The program uses synchronization mechanisms, such as mutex locks and a semaphore, to ensure safe access to shared resources.

- The log file records important events throughout the simulation.

**Methodology**

This methodology outlines the key steps and considerations for developing a train scheduling system with deadlock detection, ensuring a structured and systematic approach to the project.

1. **Problem Definition:**
   1. The project aims to simulate a train scheduling system with deadlock detection and resolution capabilities. The primary challenge is to efficiently allocate and deallocate tracks for trains, considering priorities to avoid deadlocks.
2. **Requirements Analysis:**
   1. Identify the key features and functionalities required:
   2. Train arrival simulation
   3. Resource allocation using Banker's Algorithm
   4. Deadlock detection
   5. Log generation for events
   6. User input for the number of tracks and trains
   7. User choice for entering maximum required tracks or generating random values
3. **Design:**

**Data Structures:**

* 1. Arrays to store information about tracks, trains, and resource allocation.
  2. Mutex locks for tracks, safety checks, and priority updates.
  3. Semaphore for synchronization.
  4. File pointer for logging.

**Functions:**

* 1. Modular functions for train arrival, deadlock detection, log writing, and display functions.
  2. Initialization functions for tracks and locks.
  3. User input functions for manual or random input.
  4. Thread creation and management functions.

1. **Banker's Algorithm Implementation:**
   1. Implement the Banker's Algorithm for resource allocation:
   2. Maintain arrays for maximum need, allocated, and remaining need for each train.
   3. Use mutex locks to ensure safe access to shared resources.
   4. Consider train priorities in the allocation process.
2. **Deadlock Detection and Resolution:**
   1. Implement a mechanism to periodically check for deadlocks:
   2. Use a separate thread for deadlock detection.
   3. If a deadlock is detected, release acquired locks to avoid it.
   4. Placeholder for potential deadlock resolution logic.
3. **User Interaction:**
   1. Allow users to input the number of tracks and trains.
   2. Provide options for entering maximum required tracks manually or generating random values.
4. **Logging:**
   1. Implement a logging mechanism to record important events:
   2. Log train arrivals, track allocations, deallocations, and deadlock detection.
   3. Use timestamped logs for better traceability.
5. **Multithreading:**
   1. Utilize pthread library for creating and managing threads:
   2. Threads for each train arrival.
   3. A thread for deadlock detection.
6. **Testing:**
   1. Conduct extensive testing to ensure the system's correctness and reliability:
   2. Test with different numbers of tracks and trains.
   3. Verify deadlock detection and resolution mechanisms.
   4. Check for correct resource allocation.
7. **Documentation:**
   1. Provide clear and concise documentation for the code:
   2. Explain the purpose and functionality of each function.
   3. Document the implementation of Banker's Algorithm.
   4. Detail the deadlock detection and potential resolution logic.
   5. Include information on user interactions and logging.
8. **User Guide:**
   1. Develop a user guide to assist users in running and understanding the simulation:
   2. Explain how to input values for tracks and trains.
   3. Provide instructions for manual and random input choices.
   4. Explain the log file and its significance.
9. **Deployment:**
   1. Ensure that the compiled program and any necessary dependencies are packaged appropriately for deployment.
10. **Maintenance and Future Enhancements:**
    1. Consider potential improvements and additions to the project:
    2. Enhanced deadlock resolution strategies.
    3. Graphical visualization of the train and track allocation.
11. **Feedback and Iteration:**
    1. Gather feedback from users or stakeholders.
    2. Iteratively improve the system based on feedback and identified issues.

**Code:-**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_TRACKS 10

#define MAX\_TRAINS 10

pthread\_mutex\_t track\_locks[MAX\_TRACKS];

pthread\_mutex\_t priority\_lock; // Mutex for priority updates

sem\_t semaphore;

FILE \*log\_file;

int available[MAX\_TRACKS]; // Available tracks

int max\_need[MAX\_TRAINS][MAX\_TRACKS]; // Maximum required tracks for each train

int allocated[MAX\_TRAINS][MAX\_TRACKS]; // Tracks allocated to each train

int need[MAX\_TRAINS][MAX\_TRACKS]; // Tracks still needed by each train

int finish[MAX\_TRAINS]; // Indicates if a train has finished or not

int priority[MAX\_TRAINS]; // Priority for each train

pthread\_mutex\_t safety\_check\_lock; // Lock for safety check

// Function to write log to file

void write\_log(const char \*log\_message) {

time\_t current\_time;

time(&current\_time);

// Remove newline character from asctime output

char\* time\_str = strtok(asctime(localtime(&current\_time)), "\n");

fprintf(log\_file, "[%s] %s\n", time\_str, log\_message);

fflush(log\_file);

}

void\* train\_arrival(void\* id) {

int train\_id = \*((int\*)id);

char log\_message[100];

sprintf(log\_message, "Train %d is approaching the junction with priority %d.", train\_id, priority[train\_id]);

write\_log(log\_message);

sleep(rand() % 3 + 1);

sprintf(log\_message, "Train %d has arrived at the junction.", train\_id);

write\_log(log\_message);

// Attempt to acquire tracks using Banker's Algorithm

pthread\_mutex\_lock(&safety\_check\_lock);

int safe = 0;

int deadlock\_retry = 0;

while (!safe) {

// Check if the request can be granted, considering priority

safe = 1;

for (int i = 0; i < MAX\_TRACKS; ++i) {

if (max\_need[train\_id][i] > available[i]) {

safe = 0;

break;

}

}

if (!safe) {

pthread\_mutex\_unlock(&safety\_check\_lock);

sleep(rand() % 2 + 1); // Wait before retrying request

pthread\_mutex\_lock(&safety\_check\_lock);

deadlock\_retry++;

if (deadlock\_retry >= 3) {

sprintf(log\_message, "Potential deadlock detected for Train %d. Releasing acquired locks.", train\_id);

write\_log(log\_message);

for (int i = 0; i < MAX\_TRACKS; ++i) {

available[i] += allocated[train\_id][i];

allocated[train\_id][i] = 0;

need[train\_id][i] = max\_need[train\_id][i]; // Update the 'need' array

}

pthread\_mutex\_unlock(&safety\_check\_lock);

pthread\_exit(NULL);

}

}

}

// If the request can be granted, allocate tracks

for (int i = 0; i < MAX\_TRACKS; ++i) {

allocated[train\_id][i] = max\_need[train\_id][i];

need[train\_id][i] -= max\_need[train\_id][i]; // Update the 'need' array

available[i] -= max\_need[train\_id][i];

}

pthread\_mutex\_unlock(&safety\_check\_lock);

// Simulate passing through tracks

sprintf(log\_message, "Train %d is passing through tracks.", train\_id);

write\_log(log\_message);

sleep(rand() % 3 + 2);

// Release allocated tracks

pthread\_mutex\_lock(&safety\_check\_lock);

for (int i = 0; i < MAX\_TRACKS; ++i) {

available[i] += allocated[train\_id][i];

allocated[train\_id][i] = 0;

need[train\_id][i] = max\_need[train\_id][i]; // Update the 'need' array

}

pthread\_mutex\_unlock(&safety\_check\_lock);

sprintf(log\_message, "Train %d has passed tracks.", train\_id);

write\_log(log\_message);

sprintf(log\_message, "Train %d has left the junction.", train\_id);

write\_log(log\_message);

pthread\_exit(NULL);

}

void setup\_tracks(int num\_tracks) {

for (int i = 0; i < num\_tracks; ++i) {

pthread\_mutex\_init(&track\_locks[i], NULL);

available[i] = 3; // Initializing available tracks

}

pthread\_mutex\_init(&safety\_check\_lock, NULL);

pthread\_mutex\_init(&priority\_lock, NULL); // Initialize the priority\_lock

}

void input\_max\_need(int num\_trains, int num\_tracks) {

printf("Enter the maximum required tracks for each train (0 for random values):\n");

for (int i = 0; i < num\_trains; ++i) {

printf("Train %d: ", i);

for (int j = 0; j < num\_tracks; ++j) {

scanf("%d", &max\_need[i][j]);

need[i][j] = max\_need[i][j]; // Initialize 'need' same as 'max\_need'

}

// Update priority (with mutex protection)

pthread\_mutex\_lock(&priority\_lock);

priority[i] = rand() % 2; // Assigning random priority (0 or 1)

pthread\_mutex\_unlock(&priority\_lock);

}

}

void generate\_random\_max\_need(int num\_trains, int num\_tracks) {

for (int i = 0; i < num\_trains; ++i) {

for (int j = 0; j < num\_tracks; ++j) {

max\_need[i][j] = rand() % 4 + 1; // Random values between 1 and 4

need[i][j] = max\_need[i][j]; // Initialize 'need' same as 'max\_need'

}

// Update priority (with mutex protection)

pthread\_mutex\_lock(&priority\_lock);

priority[i] = rand() % 2; // Assigning random priority (0 or 1)

pthread\_mutex\_unlock(&priority\_lock);

}

}

void display\_track\_status(int num\_tracks) {

printf("\nTrack Status:\n");

for (int i = 0; i < num\_tracks; ++i) {

printf("Track %d: Available - %d\n", i, available[i]);

}

}

void display\_train\_status(int num\_trains, int num\_tracks) {

printf("\nTrain Status:\n");

for (int i = 0; i < num\_trains; ++i) {

printf("Train %d - Need: ", i);

for (int j = 0; j < num\_tracks; ++j) {

printf("%d ", need[i][j]);

}

printf("\n");

}

}

void display\_tracks\_and\_trains(int num\_tracks, int num\_trains) {

printf("\nVisual Representation of Tracks and Trains:\n");

printf("Tracks:\n");

for (int i = 0; i < num\_tracks; ++i) {

printf("| Track %d ", i);

}

printf("|\n");

for (int i = 0; i < num\_trains; ++i) {

printf("Train %d: ", i);

for (int j = 0; j < num\_tracks; ++j) {

if (allocated[i][j] > 0) {

printf("| T%d ", i);

} else {

printf("| ");

}

}

printf("|\n");

}

}

void\* deadlock\_detection(void\* args) {

while (1) {

sleep(2); // Adjust the wait time for deadlock detection

pthread\_mutex\_lock(&safety\_check\_lock);

int deadlock\_detected = 1;

for (int i = 0; i < MAX\_TRAINS; ++i) {

if (!finish[i]) {

deadlock\_detected = 0;

break;

}

}

if (deadlock\_detected) {

printf("\nDeadlock detected! Attempting resolution...\n");

// Implement deadlock resolution logic here

printf("Deadlock resolution complete.\n");

}

pthread\_mutex\_unlock(&safety\_check\_lock);

}

}

int main() {

int num\_tracks, num\_trains;

printf("Enter the number of tracks (up to %d): ", MAX\_TRACKS);

scanf("%d", &num\_tracks);

printf("Enter the number of trains (up to %d): ", MAX\_TRAINS);

scanf("%d", &num\_trains);

if (num\_tracks > MAX\_TRACKS || num\_tracks < 1 || num\_trains > MAX\_TRAINS || num\_trains < 1) {

printf("Invalid number of tracks or trains. Exiting...\n");

return 1;

}

// Open log file for writing

log\_file = fopen("train\_simulation\_log.txt", "w");

if (log\_file == NULL) {

printf("Error opening log file. Exiting...\n");

return 1;

}

setup\_tracks(num\_tracks);

int choice;

printf("Choose option:\n1. Enter maximum required tracks manually\n2. Generate random values\nEnter choice: ");

scanf("%d", &choice);

if (choice == 1) {

input\_max\_need(num\_trains, num\_tracks);

} else if (choice == 2) {

generate\_random\_max\_need(num\_trains, num\_tracks);

} else {

printf("Invalid choice. Exiting...\n");

fclose(log\_file);

return 1;

}

pthread\_t trains[MAX\_TRAINS];

int train\_ids[MAX\_TRAINS];

srand(time(NULL));

// Creating threads for each train

for (int i = 0; i < num\_trains; ++i) {

train\_ids[i] = i;

pthread\_create(&trains[i], NULL, train\_arrival, &train\_ids[i]);

}

// Display initial track and train status

display\_track\_status(num\_tracks);

display\_train\_status(num\_trains, num\_tracks);

display\_tracks\_and\_trains(num\_tracks, num\_trains);

pthread\_t deadlock\_thread;

pthread\_create(&deadlock\_thread, NULL, deadlock\_detection, NULL);

// Waiting for all trains to finish

for (int i = 0; i < num\_trains; ++i) {

pthread\_join(trains[i], NULL);

}

// Display final track and train status

display\_track\_status(num\_tracks);

display\_train\_status(num\_trains, num\_tracks);

display\_tracks\_and\_trains(num\_tracks, num\_trains);

pthread\_cancel(deadlock\_thread); // Cancel the deadlock detection thread

// Destroying mutex locks for tracks and safety check lock

for (int i = 0; i < num\_tracks; ++i) {

pthread\_mutex\_destroy(&track\_locks[i]);

}

pthread\_mutex\_destroy(&safety\_check\_lock);

pthread\_mutex\_destroy(&priority\_lock); // Destroy the priority\_lock

// Close log file

fclose(log\_file);

return 0;

}

**Final Output:-**

* **Terminal:-**

A screenshot of a computer program

Description automatically generated

* **Log File:-**

A screenshot of a computer

Description automatically generated

**Limitations:-**

* **Simplified Simulation:**
  + The project presents a simplified simulation of train scheduling, and certain complexities of real-world railway systems may not be fully captured.
* **Placeholder Deadlock Resolution Logic:**
  + The current implementation includes a placeholder for potential deadlock resolution logic. The effectiveness of the resolution strategy depends on future development.
* **Static Train and Track Characteristics:**
  + The project assumes static characteristics for trains and tracks. In real-world scenarios, these characteristics may dynamically change, impacting scheduling dynamics.
* **Deterministic Arrival Times:**
  + Arrival times for trains are currently randomly generated, but the project does not consider dynamic variations in arrival patterns or external factors affecting train schedules.
* **Limited Resource Types:**
  + The project focuses on tracks as the primary resource, and the simulation may not generalize well to scenarios with diverse resource types and constraints.
* **Simplified Prioritization:**
  + Priority-based scheduling is implemented, but the prioritization criteria may be oversimplified. Real-world scenarios may involve more intricate prioritization factors.
* **Assumption of Predefined Maximum Tracks:**
  + The project assumes a predefined maximum number of tracks, limiting the scalability of the simulation. A more dynamic approach to track availability could enhance realism.
* **No Comprehensive Deadlock Resolution:**
  + The placeholder deadlock resolution logic is a basic implementation. In real-world scenarios, more sophisticated strategies may be needed for effective and reliable deadlock resolution.
* **Limited User Interaction Features:**
  + While the project allows users to input the number of tracks and trains and choose input methods, additional interactive features for dynamic configuration and control are limited.
* **Single Junction Focus:**
  + The simulation concentrates on a single junction. Extending the project to model interactions between multiple junctions or larger rail networks could enhance realism.

**Conclusion:-**

In conclusion, the Train Scheduling with Deadlock Detection project has successfully provided a structured and insightful exploration into the complexities of concurrent train scheduling within a railway junction. By implementing the Banker's Algorithm and prioritization strategies, the project achieves efficient resource allocation and considers the critical factor of train priorities. The integration of a deadlock detection system adds a layer of sophistication, allowing for the identification and tentative resolution of potential deadlock scenarios.

The user-friendly interface enhances adaptability, allowing users to configure the simulation according to their preferences. The project's logging system, coupled with visualizations, offers a transparent view of the dynamic allocation of tracks to trains, aiding users in understanding the system's behavior.

While the project has accomplished its primary objectives, it is essential to acknowledge certain limitations, such as the simplicity of the simulation, the static nature of train and track characteristics, and the placeholder nature of deadlock resolution logic. These limitations present opportunities for future enhancements and refinements, encouraging further exploration into realistic railway scheduling scenarios.

Overall, the Train Scheduling with Deadlock Detection project serves as a valuable learning platform, providing insights into concurrent programming, synchronization, and resource management challenges. It lays the foundation for future developments that could extend its capabilities and address more intricate aspects of real-world railway scheduling dynamics. The project's success lies not only in its functionality but also in its potential for continuous improvement and adaptation to evolving railway scheduling scenarios.

**Refrences:-**

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts. John Wiley & Sons.

2. Dijkstra, E. W. (1965). Solution of a Problem in Concurrent Programming Control. Communications of the ACM, 8(9), 569–570.

3. Tanenbaum, A. S., & Bos, H. (2014). Modern Operating Systems. Pearson.

4. Kerrisk, M. (2010). The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press.