University of Science and Technology Chittagong (USTC)
Faculty of Science, Engineering & Technology
Department of Computer Science & Engineering

Assignment - 3

Course code : CSE 328

Course Title : Operating Systems Lab

Project Title : : "Bankers Algorithm Simulator"

Submitted by :

Abhilash Chowdhury
Unique ID : 0022210005101010
Roll No : 22010110
Reg No : 888
Semester : 6th
Batch : 38th
Dept : CSE

Submitted to:

Prianka Das
[Lecturer]
Department of CSE
FSET, USTC

Abstract

The Banker's algorithm is a widely-used deadlock avoidance mechanism in operating systems. This project implements the Banker's algorithm in Python, which checks whether a system is in a safe state by allocating resources to processes dynamically. It allows users to input the allocation, maximum needed, and available resources, ensuring no deadlocks occur. The algorithm calculates the need matrix and uses a safe sequence to check if the system can safely allocate resources to all processes.

Keywords

Banker's Algorithm, Deadlock Avoidance, Resource Allocation, Safe Sequence, Need Matrix, Available Matrix, Allocation Matrix, Maximum Needed Matrix, Python.

Introduction

In operating systems, resource allocation plays a crucial role in managing processes efficiently without leading to deadlocks. The Banker's algorithm, introduced by Edsger Dijkstra, is a deadlock avoidance strategy that ensures a system is always in a safe state by checking resource allocation before granting a request. This report details the implementation of the Banker's algorithm in Python, highlighting the steps involved in determining the safe sequence and detecting deadlocks.

Background

Deadlock occurs when a group of processes is unable to proceed due to each process waiting for resources held by another. The Banker's algorithm ensures that the system remains in a safe state by simulating resource allocation requests and checking if they can be granted without leading to a deadlock. The system tracks resource allocation, available resources, and maximum required resources for each process to make informed decisions about granting or denying requests.

Banker's Algorithm Implementation

A. Algorithm Overview

The Banker's algorithm checks the system's state after each resource request to ensure that it remains in a safe state. The key components of the algorithm include:

Allocation Matrix : The current resource allocation for each process.
Maximum Needed Matrix : The maximum number of resources a process may require.
Available Matrix : The current available resources in the system.
Need Matrix : Calculated as the difference between the maximum needed and the allocated resources for each process.

B. Python Code Implementation

The project is implemented using Python, and the following sections outline the major functions:

Input Matrices : Users are prompted to enter the allocation matrix, maximum needed matrix, and available resources for all processes.

Calculate Need Matrix : The need matrix is computed by subtracting the allocation matrix from the maximum needed matrix for each process.

Display Tables : Before and after executing the safe sequence, the allocation, maximum need, remaining need, and available matrices are displayed using the PrettyTable library.

Safe Sequence Check : The algorithm checks if there is a safe sequence for executing the processes without entering a deadlock.

C. Example Usage :

Sample input for five processes and three resources:

Process	Allocation			Maximum			Available		
	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	3	2	2
P_2	2	0	0	3	2	2			
P_3	3	0	2	9	0	2			
P_4	2	1	1	4	2	2			
P_5	0	0	2	5	3	3			

Results include a safe sequence (if present) and dynamic resource tables.

Project Evaluation

The Banker's algorithm was successfully implemented using Python. The program correctly identifies whether the system is in a safe state and outputs the safe sequence if available. The tables displayed before and after the execution provide clarity on resource allocation and process status.

Setup Environment

The project was developed using Python, with the following steps involved in setting up the environment:

- Install PyCharm.
- Install Python.

- Install the **PrettyTable** library which is used to display the resource allocation and safe sequence in a tabular format.

Critical Evaluation

The Banker's Algorithm implementation works efficiently under simulated conditions, correctly identifying safe and unsafe states. However, user error during input can affect accuracy, necessitating input validation enhancements. Additionally, the implementation assumes resource availability updates occur sequentially without interruption, which may differ in real-world systems.

Conclusion

The implementation of the Banker's algorithm in Python effectively demonstrates the concepts of deadlock avoidance by ensuring resource allocation occurs only if the system remains in a safe state. This project provides valuable insights into how operating systems manage resources dynamically, preventing deadlocks through careful checking of resource allocation.

Acknowledgment

The project was completed with the guidance of Mrs. Prianka Das, whose support throughout the process was invaluable.

References

Geeks for Geeks : https://www.geeksforgeeks.org/bankers-algorithm-in-operating-system/

Javatpoint : https://www.javatpoint.com/bankers-algorithm-in-operating-system/

ResearchGate : https://www.tutorialspoint.com/banker-s-algorithm-in-operating-system/

ChatGPT : https://chatgpt.com/

Appendix

Code :

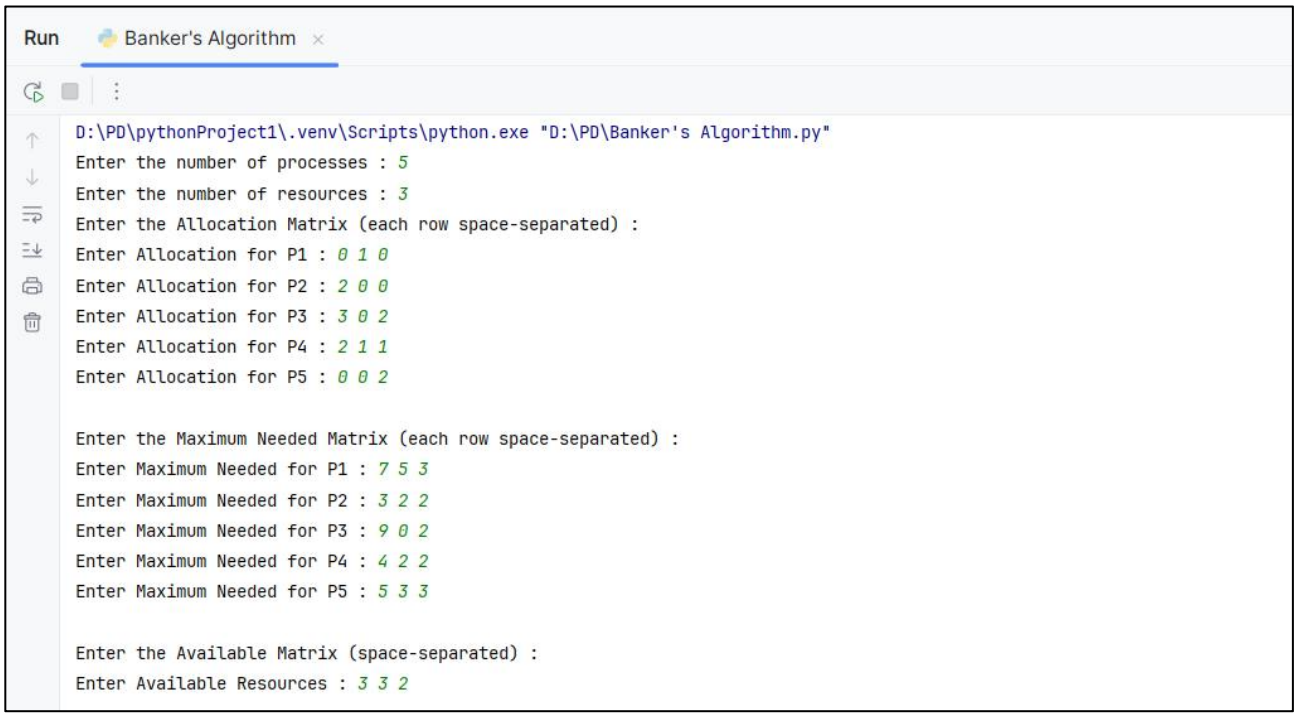
```
from prettytable import PrettyTable

class BankersAlgorithm:
    def __init__(self, num_processes, num_resources):
        self.num_processes = num_processes
        self.num_resources = num_resources
        self.process_ids = [f"P{i + 1}" for i in range(num_processes)]
        self.allocation_matrix = []
        self.max_needed_matrix = []
        self.available_matrix = []
        self.need_matrix = []
        self.finish = [False] * num_processes
```

```
    def input_matrices(self):
        print("Enter the Allocation Matrix (each row space-separated) :")
        for i in range(self.num_processes):
            allocation = list(map(int, input(f"Enter Allocation for {self.process_ids[i]} : ").split()))
            self.allocation_matrix.append(allocation)

        print("\nEnter the Maximum Needed Matrix (each row space-separated) :")
        for i in range(self.num_processes):
            max_needed = list(map(int, input(f"Enter Maximum Needed for {self.process_ids[i]} :
").split()))
            self.max_needed_matrix.append(max_needed)

        print("\nEnter the Available Matrix (space-separated) :")
        self.available_matrix = list(map(int, input("Enter Available Resources : ").split()))

        self.calculate_need_matrix()

    def calculate_need_matrix(self):
        self.need_matrix = [[self.max_needed_matrix[i][j] - self.allocation_matrix[i][j]
                             for j in range(self.num_resources)] for i in range(self.num_processes)]

    def display_table_before_execution(self):
        table = PrettyTable()
        headers = ["Process", "Allocation", "Maximum Need", "Available"]
        table.field_names = headers

        for idx, process in enumerate(self.process_ids):
            allocation = ' '.join(map(str, self.allocation_matrix[idx]))
            max_need = ' '.join(map(str, self.max_needed_matrix[idx]))

            available_str = ' '.join(map(str, self.available_matrix)) if idx == 0 else " "

            table.add_row([self.process_ids[idx], allocation, max_need, available_str])

        print("\nResource Allocation Table :")
        print(table)

    def display_table_after_execution(self, sequence):
        table = PrettyTable()
        headers = ["Process", "Allocation", "Maximum Need", "Remaining Need", "Available"]
        table.field_names = headers

        work = self.available_matrix[:]
        for process in sequence:
            idx = self.process_ids.index(process)
            allocation = ' '.join(map(str, self.allocation_matrix[idx]))
            max_need = ' '.join(map(str, self.max_needed_matrix[idx]))
            remaining_need = ' '.join(map(str, self.need_matrix[idx]))
```

```
            work = [work[j] + self.allocation_matrix[idx][j] for j in range(self.num_resources)]
            available_str = " ".join(map(str, work))

            table.add_row([self.process_ids[idx], allocation, max_need, remaining_need, available_str])

        print("\nResource Allocation Table (After Safe Sequence Execution):")
        print(table)

    def is_safe(self):
        work = self.available_matrix[:]
        safe_sequence = []

        while len(safe_sequence) < self.num_processes:
            progress_made = False
            for i in range(self.num_processes):
                if not self.finish[i] and all(self.need_matrix[i][j] <= work[j] for j in
range(self.num_resources)):
                    safe_sequence.append(self.process_ids[i])
                    work = [work[j] + self.allocation_matrix[i][j] for j in
                        range(self.num_resources)]
                    self.finish[i] = True
                    progress_made = True
                    break
            if not progress_made:
                return False, []
        return True, safe_sequence

    def detect_deadlock(self):
        is_safe, safe_sequence = self.is_safe()

        if is_safe:
            self.display_table_before_execution()
            print("\nThe system is in a safe state.")
            self.display_table_after_execution(safe_sequence)
            print("\nSafe Sequence :", " -> ".join(safe_sequence))
        else:
            self.display_table_before_execution()
            print("\nThe system is in a deadlock state.")
            print("\nNo Safe Sequence Exists.")

if __name__ == "__main__":
    num_processes = int(input("Enter the number of processes : "))
    num_resources = int(input("Enter the number of resources : "))

    bankers_algo = BankersAlgorithm(num_processes, num_resources)
    bankers_algo.input_matrices()
    bankers_algo.detect_deadlock()
```

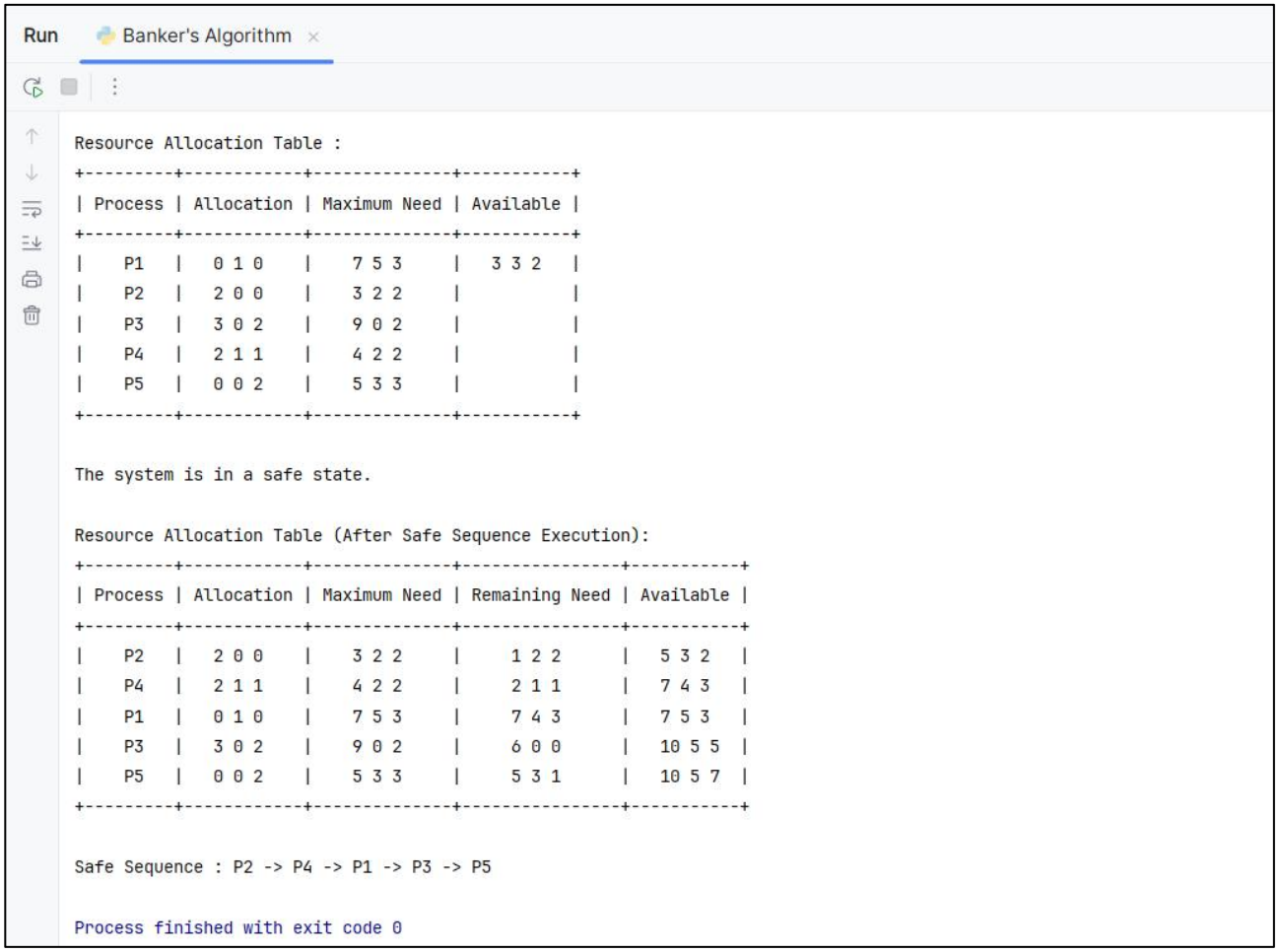
Example : If the system is in a Safe State

Input :



```
Run  Banker's Algorithm
D:\PD\pythonProject1\.venv\Scripts\python.exe "D:\PD\Banker's Algorithm.py"
Enter the number of processes : 5
Enter the number of resources : 3
Enter the Allocation Matrix (each row space-separated) :
Enter Allocation for P1 : 0 1 0
Enter Allocation for P2 : 2 0 0
Enter Allocation for P3 : 3 0 2
Enter Allocation for P4 : 2 1 1
Enter Allocation for P5 : 0 0 2

Enter the Maximum Needed Matrix (each row space-separated) :
Enter Maximum Needed for P1 : 7 5 3
Enter Maximum Needed for P2 : 3 2 2
Enter Maximum Needed for P3 : 9 0 2
Enter Maximum Needed for P4 : 4 2 2
Enter Maximum Needed for P5 : 5 3 3

Enter the Available Matrix (space-separated) :
Enter Available Resources : 3 3 2
```

Output :



```
Run  Banker's Algorithm
Resource Allocation Table :
+---------+------------+--------------+-----------+
| Process | Allocation | Maximum Need | Available |
+---------+------------+--------------+-----------+
|    P1   |   0 1 0    |    7 5 3     |   3 3 2   |
|    P2   |   2 0 0    |    3 2 2     |           |
|    P3   |   3 0 2    |    9 0 2     |           |
|    P4   |   2 1 1    |    4 2 2     |           |
|    P5   |   0 0 2    |    5 3 3     |           |
+---------+------------+--------------+-----------+

The system is in a safe state.

Resource Allocation Table (After Safe Sequence Execution):
+---------+------------+--------------+----------------+-----------+
| Process | Allocation | Maximum Need | Remaining Need | Available |
+---------+------------+--------------+----------------+-----------+
|    P2   |   2 0 0    |    3 2 2     |     1 2 2      |   5 3 2   |
|    P4   |   2 1 1    |    4 2 2     |     2 1 1      |   7 4 3   |
|    P1   |   0 1 0    |    7 5 3     |     7 4 3      |   7 5 3   |
|    P3   |   3 0 2    |    9 0 2     |     6 0 0      |   10 5 5  |
|    P5   |   0 0 2    |    5 3 3     |     5 3 1      |   10 5 7  |
+---------+------------+--------------+----------------+-----------+

Safe Sequence : P2 -> P4 -> P1 -> P3 -> P5

Process finished with exit code 0
```

Example : If the system is in a Deadlock State

Input :

```
Run  Banker's Algorithm

D:\PD\pythonProject1\.venv\Scripts\python.exe "D:\PD\Banker's Algorithm.py"
Enter the number of processes : 5
Enter the number of resources : 3
Enter the Allocation Matrix (each row space-separated) :
Enter Allocation for P1 : 2 1 1
Enter Allocation for P2 : 1 2 1
Enter Allocation for P3 : 2 2 2
Enter Allocation for P4 : 1 1 1
Enter Allocation for P5 : 0 1 1

Enter the Maximum Needed Matrix (each row space-separated) :
Enter Maximum Needed for P1 : 4 3 31
Enter Maximum Needed for P2 : 2 2 2
Enter Maximum Needed for P3 : 4 3 3
Enter Maximum Needed for P4 : 2 2 2
Enter Maximum Needed for P5 : 1 2 2

Enter the Available Matrix (space-separated) :
Enter Available Resources : 2 2 2
```

Output :

```
Run  Banker's Algorithm

Resource Allocation Table :
+---------+------------+--------------+-----------+
| Process | Allocation | Maximum Need | Available |
+---------+------------+--------------+-----------+
|    P1   |   2 1 1    |    4 3 31    |   2 2 2   |
|    P2   |   1 2 1    |    2 2 2     |           |
|    P3   |   2 2 2    |    4 3 3     |           |
|    P4   |   1 1 1    |    2 2 2     |           |
|    P5   |   0 1 1    |    1 2 2     |           |
+---------+------------+--------------+-----------+

The system is in a deadlock state.

No Safe Sequence Exists.

Process finished with exit code 0
```