TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A Lab Report

of

Distributed Systems

on

Implementation of Banker's Algorithm for Avoiding Deadlock

Submitted by:

Nistha Bajracharya

THA076BCT024

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

August, 2023

TITLE: IMPLEMENTATION OF BANKER'S ALGORITHM FOR AVOIDING DEADLOCK

THEORY:

Banker's Algorithm:

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems to manage resource requests made by multiple processes running concurrently. It is designed to prevent deadlocks, which occur when processes are waiting for resources held by other processes, leading to a situation where none of the processes can proceed.

The Banker's Algorithm was proposed by Edsger W. Dijkstra in 1965. It is based on the concept of a banking system, where resources are treated as "currency" that can be allocated to processes. The algorithm allows resources to be allocated to processes in a way that ensures the system's safety, meaning that it can always find a safe sequence of resource allocations that avoids deadlock.

The Banker's Algorithm works as follows:

1. Initially, the system has a fixed number of resources (e.g., CPU, memory, printers) and a fixed number of instances for each resource type.
2. Each process in the system must declare the maximum number of instances of each resource it may need during its execution (Max Claim) and the currently allocated resources (Allocated).
3. Based on the Max Claim and Allocated information, the algorithm calculates the resource need for each process (Needed = Max Claim - Allocated).
4. When a process requests additional resources, the system checks if the requested resources can be granted without leading to an unsafe state (possible deadlock). It does this by simulating the allocation and checking if a safe sequence exists.
5. If granting the requested resources would lead to an unsafe state, the system denies the request, and the process must wait until it can obtain the necessary resources without causing deadlock.

6. If granting the requested resources would not lead to an unsafe state, the system temporarily allocates the resources to the process and checks again if the system remains in a safe state. If the system remains safe, the resources are permanently allocated to the process; otherwise, the temporary allocation is reverted.

CODE

```
import numpy as np

def is_safe_to_allocate(process_index):
    for resource_index in range(num_resources):
        if needed[process_index][resource_index] > available[resource_index]:
            return False
    return True

num_processes = 5
num_resources = 4

# The sequence in which processes are allocated
safe_sequence = np.zeros((num_processes,), dtype=int)

# A flag to keep track of visited processes
visited = np.zeros((num_processes,), dtype=int)

# Allocated resources for each process
allocated = np.array([[4, 0, 0, 1],
                      [1, 1, 0, 0],
                      [1, 2, 5, 4],
                      [0, 6, 3, 3],
                      [0, 2, 1, 2]])

# Maximum required resources for each process
maximum = np.array([[6, 0, 1, 2],
                    [1, 7, 5, 0],
                    [2, 3, 5, 6],
                    [1, 6, 5, 3],
                    [1, 6, 5, 6]])

# Calculate the needed resources for each process
needed = maximum - allocated

# Available resources in the system
available = np.array([3, 2, 1, 1])
```

```
count = 0
while count < num_processes:
    is_allocatable = False
    for process_index in range(num_processes):
        if visited[process_index] == 0:
            if is_safe_to_allocate(process_index):
                # Allocate resources to the process
                safe_sequence[count] = process_index
                count += 1
                visited[process_index] = 1
                is_allocatable = True
                # Release resources back to the system after the process finishes
                for resource_index in range(num_resources):
                    available[resource_index] += allocated[process_index][resource_index]
    if not is_allocatable:
        break

if count < num_processes:
    print('The system is Unsafe')
else:
    print("The system is Safe")
    print("Safe Sequence: ", safe_sequence)
    print("Available Resources:", available)
```

RESULT

```
The system is Safe
Safe Sequence:  [0 2 3 4 1]
Available Resources: [ 9 13 10 11]
```

DISCUSSION

In this lab, we implemented the Banker's algorithm in Python. We tested the algorithm on a number of different examples, and we verified that the algorithm was able to prevent deadlocks from occurring. The Banker's algorithm is a relatively simple algorithm to implement, and it is often used in operating systems.

CONCLUSION

The implementation of the Banker's Algorithm provides a robust resource allocation and deadlock avoidance mechanism for concurrent processes in modern operating systems. By carefully managing resource requests and checking for safe sequences of allocations, the algorithm ensures that processes can execute without the risk of deadlock. It efficiently manages available resources, dynamically adjusts allocations, and guarantees system stability by preventing unsafe resource states. With its real-world applicability and support for multiple concurrent processes, the Banker's Algorithm enhances the stability and efficiency of operating systems, facilitating seamless execution and resource sharing.