
**TRIBHUVAN UNIVERSITY**

**INSTITUTE OF ENGINEERING**

**THAPATHALI CAMPUS**

**A Lab Report**

**Of**

**Distributed System**

**On**

# Implementation of Banker's Algorithm for Avoiding Deadlock

**Submitted By:**

**Santosh Pandey (THA076BCT041)**

**Submitted To:**

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

6th August, 2023

# TITLE: IMPLEMENTATION OF BANKER'S ALGORITHM FOR AVOIDING DEADLOCK

## THEORY:

### Banker's Algorithm

The Banker's Algorithm, created by Edsger W. Dijkstra in 1965, is a resource allocation and deadlock avoidance algorithm in operating systems. Its main goal is to manage resource allocation to multiple processes to maintain a safe system state and prevent deadlock.

### Basic Concepts

1. **Resources**: Resources in a computer system refer to various elements necessary for a process to complete its execution, such as CPU time, memory, files, and I/O devices.
2. **Available Resources**: The resources currently accessible and ready to be allocated to processes.
3. **Maximum Need**: The highest number of resources a process might require during its execution.
4. **Allocated Resources**: The resources that are currently allocated to each process.
5. **Need**: Need signifies the difference between the maximum need and the presently allocated resources for each process.

### Deadlock

Deadlock is a condition in a computer system where processes are mutually waiting for resources held by each other, resulting in a state of inactivity. In such a situation, the processes are effectively blocked, leading to a lack of progress and potential performance issues or a complete system freeze.

### Steps to Implement Banker's Algorithm:

The Banker's Algorithm works by simulating resource allocation requests and checking if the system can enter a safe state. A safe state is a state in which there is a sequence of resource allocation that allows all processes to complete their execution without causing deadlock. The

algorithm proceeds as follows:

- **Gather Information:** Collect data about the system's resources, including the number of resource types, the maximum available resources of each type, and the current allocation and maximum need of each process.
- **Calculate Available Resources**: Calculate the available resources for each resource type by subtracting the total allocated resources from the maximum resources.
- **Initialize Data Structures:** Set up data structures to track the state of the system, such as arrays to store the maximum need, allocated resources, and need of each process.
- **Check for Safe State:** Perform a safety check to ensure that the system is currently in a safe state and no deadlock exists. Simulate resource allocation for each process and verify if all processes can complete without getting stuck.
- **Request Handling:** When a process requests resources, check if the requested amount is valid (within its maximum need) and if the requested resources can be allocated based on the available resources. If the request is granted, update the data structures accordingly.
- **Check for Deadlock:** After resource allocation or request handling, check if the system falls into a deadlock state. If so, take appropriate action, such as rolling back the resource allocation or denying the request.
- **Resource Release:** When a process completes its execution, release the allocated resources and update the available resources.
- **Repeat:** Continue monitoring resource requests and releases, repeating steps 4 to 7 to ensure the system remains in a safe state and to avoid deadlocks.

## CODE:

Let's consider a simple example where we can implement Banker's Algorithm for avoiding Deadlock:

```
class BankerAlgorithm:
    def __init__(self, total_resources, total_processes):
        # Initialize the Banker's Algorithm with the total number of resources
and processes
        self.total_resources = total_resources
        self.total_processes = total_processes
        self.available = total_resources.copy()
        self.maximum = [[0 for _ in range(len(total_resources))] for _ in
range(total_processes)]
        self.allocated = [[0 for _ in range(len(total_resources))] for _ in
range(total_processes)]
        self.need = [[0 for _ in range(len(total_resources))] for _ in
range(total_processes)]
        self.finish = [False] * total_processes

    def set_maximum_resources(self, max_resources):
        # Set the maximum need of each process
        for i in range(self.total_processes):
            for j in range(len(max_resources[i])):
                self.maximum[i][j] = max_resources[i][j]
                self.need[i][j] = max_resources[i][j] - self.allocated[i][j]

    def set_allocated_resources(self, allocated_resources):
        # Set the allocated resources of each process
        for i in range(self.total_processes):
            for j in range(len(allocated_resources[i])):
                self.allocated[i][j] = allocated_resources[i][j]
                self.available[j] -= allocated_resources[i][j]
                self.need[i][j] = self.maximum[i][j] - self.allocated[i][j]

    def request_resources(self, process_id, requested_resources):
        # Check if the requested resources can be granted while keeping the
system in a safe state
        for i in range(len(requested_resources)):
            if requested_resources[i] > self.need[process_id][i]:
                print("Error: Process has exceeded its maximum claim.")
                return False
```

```
            if requested_resources[i] > self.available[i]:
                print("Process must wait. Resources are not available.")
                return False

        # Temporary allocation to check for safety
        for i in range(len(requested_resources)):
            self.available[i] -= requested_resources[i]
            self.allocated[process_id][i] += requested_resources[i]
            self.need[process_id][i] -= requested_resources[i]

        if self.is_safe_state():
            print("Resources allocated to Process", process_id)
            return True
        else:
            print("Resources denied for Process", process_id, "to avoid
deadlock.")
            # Roll back the temporary allocation
            for i in range(len(requested_resources)):
                self.available[i] += requested_resources[i]
                self.allocated[process_id][i] -= requested_resources[i]
                self.need[process_id][i] += requested_resources[i]
            return False

    def release_resources(self, process_id):
        # Release the resources held by the specified process
        for i in range(len(self.allocated[process_id])):
            self.available[i] += self.allocated[process_id][i]
            self.allocated[process_id][i] = 0
            self.need[process_id][i] = self.maximum[process_id][i]

    def is_safe_state(self):
        # Check if the current system state is safe (no deadlock)
        work = self.available.copy()
        self.finish = [False] * self.total_processes
        safe_sequence = []

        while True:
            found = False
            for i in range(self.total_processes):
                if not self.finish[i] and all(self.need[i][j] <= work[j] for j in
range(len(work))):
                    work = [work[j] + self.allocated[i][j] for j in
range(len(work))]
                    self.finish[i] = True
                    safe_sequence.append(i)
```

```
                    found = True

            if not found:
                break

        if all(self.finish):
            print("Safe Sequence:", safe_sequence)
            return True
        else:
            print("System is in an unsafe state (deadlock).")
            return False


# Example usage
total_resources = [8, 6, 9]
total_processes = 6
banker_algorithm = BankerAlgorithm(total_resources, total_processes)

maximum_resources = [
    [4, 3, 2],
    [1, 2, 2],
    [7, 0, 5],
    [0, 1, 1],
    [4, 3, 1],
    [3, 2, 1]
]

allocated_resources = [
    [1, 0, 0],
    [0, 1, 1],
    [3, 0, 2],
    [2, 1, 1],
    [0, 0, 2],
    [0, 0, 0]
]

banker_algorithm.set_maximum_resources(maximum_resources)
banker_algorithm.set_allocated_resources(allocated_resources)

if banker_algorithm.is_safe_state():
    # Resource request for Process 5
    process_id = 5
    requested_resources = [1, 1, 0]
    banker_algorithm.request_resources(process_id, requested_resources)
```

```
    # Resource release for Process 2
    process_id = 2
    banker_algorithm.release_resources(process_id)
```

**OUTPUT:**

```
lab4>python "Banker's Algorithm.py"
Safe Sequence: [1, 3, 4, 5, 0, 2]
Safe Sequence: [1, 3, 5, 0, 2, 4]
Resources allocated to Process 5
```

**DISCUSSION:**

The implementation of the Banker's Algorithm offers a robust approach to manage resource allocation and prevent deadlock in multi-process systems. By simulating resource requests and ensuring system safety, it effectively avoids deadlock situations. The utilization of arrays for tracking resource availability, maximum needs, and allocations simplifies resource management. Furthermore, the safety check algorithm ensures a secure sequence of processes, making the system stable and efficient.

**CONCLUSION:**

The Banker's Algorithm is a potent resource allocation and deadlock avoidance mechanism in sophisticated operating systems. Its dynamic safety checks before granting resource requests guarantee system stability. By utilizing arrays and data structures, the implementation simplifies resource tracking and management. When used correctly, the Banker's Algorithm enhances the efficiency of multi-process environments, offering a dependable solution to resource allocation and enabling smooth process execution without encountering deadlock.