INDEX

OBJECTIVE

Ensuring System Stability: The primary goal is to maintain the stability and reliability of the operating system by avoiding deadlocks. Deadlocks can lead to system crashes or processes becoming unresponsive, and the Banker's algorithm helps to prevent such scenarios.

Resource Allocation: The algorithm assists in allocating resources to processes in a manner that avoids potential deadlocks. By monitoring the resource allocation requests and ensuring that they do not lead to an unsafe state, the algorithm prevents deadlock conditions.

Optimizing Resource Utilization: While preventing deadlocks, the Banker's algorithm also aims to optimize resource utilization. It ensures that resources are allocated to processes in such a way that they are efficiently utilized without causing a deadlock.

Predictable System Behavior: With the Banker's algorithm in place, the system's behavior becomes more predictable. Processes can request and release resources, and the algorithm's checks help in making informed decisions about resource allocation, thereby reducing the likelihood of deadlock.

Fair Resource Allocation: The algorithm helps in achieving a fair distribution of resources among competing processes. It ensures that no process is unfairly starved of resources, which could potentially lead to deadlock situations.

Real-time Systems: For real-time systems, preventing deadlocks is crucial to meet deadlines and ensure system responsiveness. The Banker's algorithm helps in guaranteeing that deadlines are met by preventing deadlocks that could disrupt the execution of critical processes.

Enhanced System Efficiency: By avoiding deadlock scenarios, the system becomes more efficient as resources are used optimally, and there is no unnecessary waiting or blocking of processes due to resource unavailability.

Reduced Administrative Overhead: The algorithm reduces the need for manual intervention in resource allocation decisions. This leads to a reduction in administrative overhead and automates the process of preventing deadlocks.

Improved User Experience: Users of the system experience a smoother and more reliable environment. They can trust that their applications and processes will run without being affected by deadlock-related issues.

Compliance with Service Level Agreements (SLAs): In cases where the operating system serves as a platform for hosting services or applications, preventing deadlocks is critical to meeting SLAs and ensuring a high level of service availability.

PROBLEM STATEMENT

Problem Overview:

The objective of this project is to design, implement, and integrate a deadlock prevention mechanism based on the Banker's Algorithm into an operating system. The system should effectively identify and prevent potential deadlocks, ensuring that the system remains responsive and reliable.

System Requirements:

To accomplish the goal, the system should fulfill the following requirements:

a. Resource Allocation Model: Implement a resource allocation model that tracks the allocation of resources to processes, the maximum resources that each process can request, and the current available resources.

b. Deadlock Detection: Develop a mechanism to detect when a deadlock is imminent, based on the resource allocation model.

c. Deadlock Prevention: Implement the Banker's Algorithm as a preemptive strategy to avoid the occurrence of deadlocks.

d. Resource Allocation and Deallocation: Allow processes to request resources and release them when they are no longer needed. The system should ensure that resources are allocated only if the allocation is safe to prevent deadlocks.

e. User Interface: Provide a user-friendly interface to allow system administrators to monitor and manage resource allocation and view the status of processes.

System Functionality:

The proposed system should operate as follows:

a. On system startup, initialize the resource allocation model with available resources and maximum resource claims for each process.

b. As processes request resources, the system should analyze these requests using the Banker's Algorithm to determine if granting the request will lead to a safe state. If so, allocate the resources; otherwise, block the process.

c. Processes that no longer need resources should release them, making them available for allocation to other processes.

d. Periodically check the resource allocation model to detect potential deadlocks. If a deadlock is detected, the system should take action to resolve it.

Evaluation and Testing:

The system's effectiveness should be evaluated through extensive testing, including the use of simulated scenarios to assess its ability to prevent deadlocks. The system should demonstrate its capacity to maintain system stability even under heavy resource contention.

Documentation:

Comprehensive documentation, including user manuals and technical documentation, should be provided to aid system administrators in understanding and operating the deadlock prevention system.

Implementation:

The system should be implemented in an existing operating system or developed as an independent tool that can be integrated into various operating systems.

Security and Robustness:

Ensure the system's security by preventing unauthorized access to resource allocation and maintaining the integrity of the resource allocation model. Additionally, the system should be robust and handle various system loads without compromising performance or safety.

IMPLEMENTATION

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm that is used in operating systems to prevent deadlocks. The algorithm was proposed by Edsger Dijkstra in 1965.

The Banker's Algorithm works by maintaining a system state table that tracks the current allocation of resources to each process, the maximum possible allocation of resources to each process, and the total available resources in the system. The algorithm also tracks the number of claims made by each process, which is the number of resources that the process has requested but has not yet been allocated.

Whenever a process requests a resource, the Banker's Algorithm checks to see if the request can be granted without causing a deadlock. If the request can be granted, the algorithm allocates the resource to the process and updates the system state table. If the request cannot be granted without causing a deadlock, the algorithm denies the request and the process must wait until the resources become available.

The Banker's Algorithm is a very effective way to prevent deadlocks, but it can be complex to implement. The following steps outline how to implement the Banker's Algorithm in an operating system:

Create a system state table that tracks the current allocation of resources to each process, the maximum possible allocation of resources to each process, and the total available resources in the system.

When a process requests a resource, check to see if the request can be granted without causing a deadlock.

If the request can be granted, allocate the resource to the process and update the system state table.

If the request cannot be granted without causing a deadlock, deny the request and the process must wait until the resources become available.

CODE IMPLEMENTATION

```
#include <iostream>
using namespace std;

int main() {
   int instance[5], count, sequence[10], safe, s = 0, j, completed, i;
   int available[5], allocation[10][5], max[10][5];
   int need[10][5], process, P[10], countofr, countofp, running[10];

   cout << "\nEnter the number of resources (<=5): ";
   cin >> countofr;

   for (int i = 0; i < countofr; i++) {
      cout << "\nEnter the max instances of resource R[" << i << "]: ";
      cin >> instance[i];
      available[i] = instance[i];
   }

   cout << "\nEnter the number of processes (<=10): ";
   cin >> countofp;

   cout << "\nEnter the allocation matrix: \n";
   for (i = 0; i < countofp; i++) {
      cout << "FOR THE PROCESS: P[" << i << "]" << endl;
      for (int j = 0; j < countofr; j++) {
         cout << "Allocation of resource R[" << j << "] is: ";
         cin >> allocation[i][j];
         available[j] -= allocation[i][j];
```

```
    }
}

cout << "\nEnter the MAX matrix:\n";
for (i = 0; i < countofp; i++) {
    cout << "FOR THE PROCESS P[" << i << "]" << endl;
    for (int j = 0; j < countofr; j++) {
        cout << "Max demand of resource R[" << j << "] is: ";
        cin >> max[i][j];
    }
}

cout << "\nThe given data are: \n";
cout << endl << "\nTotal resources in system: \n\n ";
for (i = 0; i < countofr; i++)
    cout << "R[" << i << "]";
cout << endl;
for (i = 0; i < countofr; i++)
    cout << " " << instance[i];
cout << "\n\n ALLOCATION matrix \n\n\t";
for (j = 0; j < countofr; j++) {
    cout << "R[" << j << "]";
}
cout << endl;

for (i = 0; i < countofp; i++) {
    cout << "P[" << i << "] ";
    for (j = 0; j < countofr; j++)
        cout << " " << allocation[i][j];
```

```
    cout << endl;
}

for (i = 0; i < countofp; i++) {
    for (j = 0; j < countofr; j++) {
        need[i][j] = max[i][j] - allocation[i][j];
    }
}

cout << "\n\n NEED matrix \n\n\t";
for (j = 0; j < countofr; j++) {
    cout << "R[" << j << "]";
}
cout << endl;

for (i = 0; i < countofp; i++) {
    cout << "P[" << i << "]";
    for (j = 0; j < countofr; j++)
        cout << " " << need[i][j];
    cout << endl;
}

cout << "\nNow to check whether the above state is safe";
cout << "\nSequence in which above requests can be fulfilled";
cout << "\nPress any key to continue";
cin.ignore();
cin.get();

count = countofp;
```

```
for (i = 0; i < countofp; i++) {
   running[i] = 1;
}

while (count) {
   safe = 0;
   for (i = 0; i < countofp; i++) {
      if (running[i]) {
         completed = 1;
         for (j = 0; j < countofr; j++)
            if (need[i][j] > available[j])
               completed = 0;
         if (completed) {
            running[i] = 0;
            count--;
            safe = 1;
            for (j = 0; j < countofr; j++)
               available[j] += allocation[i][j];
            sequence[s++] = i;
            cout << "\nRunning process P[" << i << "]";
            cout << endl << "\nTotal resources now available:\n\n";
            for (int k = 0; k < countofr; k++)
               cout << "R[" << k << "]";
            cout << endl;
            for (int k = 0; k < countofr; k++)
               cout << " " << available[k];
            break;
         }
```

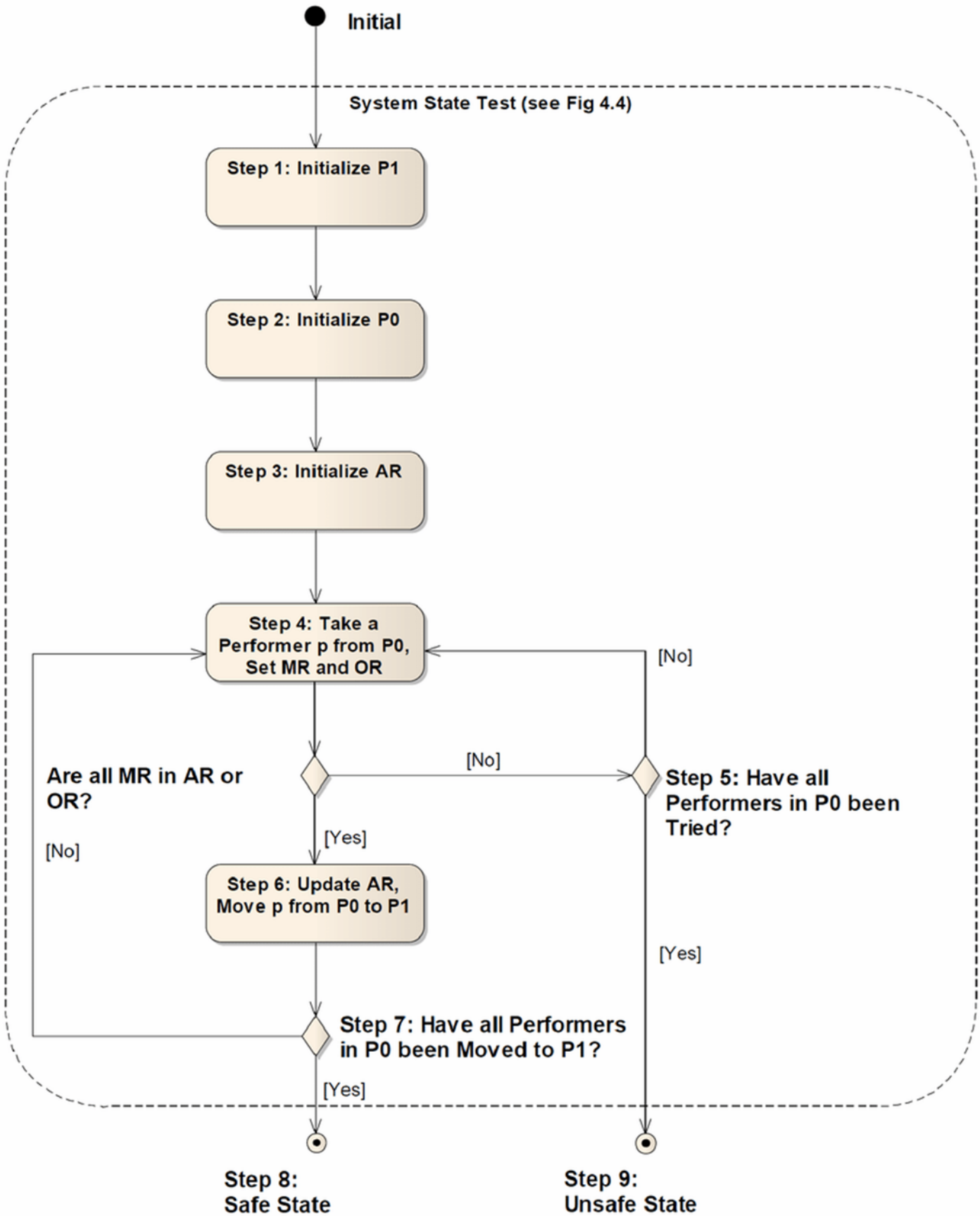
```
            }
        }
        if (!safe)
            break;
    }

    if (safe) {
        cout << "\nThe System is in a safe state";
        cout << "\nSafe sequence is :";
        for (i = 0; i < countofp; i++)
            cout << "\tP[" << sequence[i] << "]";
    } else {
        cout << "\nThe System is in an unsafe state";
    }

    cin.ignore();
    cin.get();

    return 0;
}
```

OUTPUT SCREENSHOTS

```
Enter the number of resources (<=5): 2

Enter the max instances of resource R[0]: 2

Enter the max instances of resource R[1]: 2

Enter the number of processes (<=10): 3

Enter the allocation matrix:
FOR THE PROCESS: P[0]
Allocation of resource R[0] is: 2
Allocation of resource R[1] is: 2
FOR THE PROCESS: P[1]
Allocation of resource R[0] is: 1
Allocation of resource R[1] is: 3
FOR THE PROCESS: P[2]
Allocation of resource R[0] is: 2
Allocation of resource R[1] is: 1

Enter the MAX matrix:
FOR THE PROCESS P[0]
Max demand of resource R[0] is: 3
Max demand of resource R[1] is: 3
FOR THE PROCESS P[1]
Max demand of resource R[0] is: 3
Max demand of resource R[1] is: 3
FOR THE PROCESS P[2]
Max demand of resource R[0] is: 2
Max demand of resource R[1] is: 3
```

```
The given data are:


Total resources in system:

 R[0]R[1]
 2 2

 ALLOCATION matrix

        R[0]R[1]
P[0]  2 2
P[1]  1 3
P[2]  2 1


 NEED matrix

        R[0]R[1]
P[0] 1 1
P[1] 2 0
P[2] 0 2

Now to check whether the above state is safe
Sequence in which above requests can be fulfilled
Press any key to continue
The System is in an unsafe state
```

ARCHITECTURE DIAGRAM



RESULT

Banker's Algorithm is a deadlock avoidance algorithm that works by tracking the allocation of resources and preventing processes from requesting more resources than are available.This algorithm helps to ensure that all processes have the resources they need to complete their tasks without causing deadlocks.Banker's Algorithm can be complex to implement, but it is a very effective way to prevent deadlocks in operating systems.

REFERENCES

62757735

"Formal Verification of Deadlock Avoidance Protocols in Communication Networks"	I. Lee, J. de Oliveira, K. Obraczka	2009	Focuses on formal verification of deadlock avoidance protocols in communication networks.
"Deadlock Detection in Wireless Sensor Networks: A Comprehensive Survey"	S. Ahmed, S. S. Kanhere	2010	Reviews deadlock detection in wireless sensor networks, an emerging area.
"Modeling and Analysis of Deadlocks in Robot Manipulators"	M. Pradeep, P. S. Gandhi	2012	Discusses deadlock modeling and analysis in robot manipulators.
"A Review on Deadlock Handling in Wireless Sensor Networks"	A. K. Gupta, D. S. Rajpoot	2013	Provides insights into deadlock handling strategies in wireless sensor networks.
"Deadlock Detection and Resolution in Parallel and Distributed Systems"	K. Agarwal, R. Gupta	2015	Explores deadlock detection and resolution in parallel and distributed computing environments.
"Formal Verification of Deadlock Freedom for Concurrent Programs"	M. Abadi, C. Fournet	2017	Discusses formal verification methods for ensuring deadlock freedom in concurrent programs.
"A Survey of Deadlock Detection and Resolution Techniques in Multithreading Environments"	S. J. Moon, D. B. Lee	2020	Examines deadlock detection and resolution techniques in multithreading environments.

CONCLUSION

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm that is used in operating systems to prevent deadlocks. It works by maintaining a system state table that tracks the current allocation of resources to each process, the maximum possible allocation of resources to each process, and the total available resources in the system. The algorithm also tracks the number of claims made by each process, which is the number of resources that the process has requested but has not yet been allocated.

The Banker's Algorithm is very effective at preventing deadlocks, but it can be complex to implement. It is also important to note that the Banker's Algorithm is only a deadlock avoidance algorithm, and it cannot detect or resolve deadlocks that have already occurred.

Here are some of the advantages of using the Banker's Algorithm:

It is very effective at preventing deadlocks.

It is relatively easy to understand and implement.

It can be used in a wide variety of operating systems.

Here are some of the disadvantages of using the Banker's Algorithm:

It can be complex to implement.

It is only a deadlock avoidance algorithm, and it cannot detect or resolve deadlocks that have already occurred.

It can lead to resource underutilization, as it may deny requests even when there are enough resources available to satisfy them.

Overall, the Banker's Algorithm is a very effective way to prevent deadlocks in operating systems. It is important to weigh the advantages and disadvantages of using the algorithm before deciding whether to implement it.

TEAM MEMBERS:

RA2211003011413-SAI NAGA SURYA BOYINA

RA2211003011414-JASWANTH MARNI

RA2211003011456-AKSHAY VARUN MAKA