Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.

Name: Adwait S Purao

UID: 2021300101

Batch: B2

Experiment no.: 7

Aim: To implement banker's algorithm

Banker's Algorithm in Operating System (OS)

It is a banker algorithm used to **avoid deadlock** and **allocate resources** safely to each process in the computer system. The '**S-State'** examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes. The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources. In this section, we will learn the **Banker's Algorithm** in detail. Also, we will solve problems based on the **Banker's Algorithm**. To understand the Banker's Algorithm first we will see a real word example of it.

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system.

Similarly, it works in an **operating system**. When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as **deadlock avoidance algorithm** or **deadlock detection** in the operating system.

Advantages

Following are the essential characteristics of the Banker's algorithm:

1. It contains various resources that meet the requirements of each process.
2. Each process should provide information to the operating system for upcoming resource requests, the number of resources, and how long the resources will be held.
3. It helps the operating system manage and control process requests for each type of resource in the computer system.
4. The algorithm has a Max resource attribute that represents indicates each process can hold the maximum number of resources in a system.

Disadvantages

1. It requires a fixed number of processes, and no additional processes can be started in the system while executing the process.
2. The algorithm does no longer allows the processes to exchange its maximum needs while processing its tasks.
3. Each process has to know and state their maximum resource requirement in advance for the system.
4. The number of resource requests can be granted in a finite time, but the time limit for allocating the resources is one year.

When working with a banker's algorithm, it requests to know about three things:

1. How much each process can request for each resource in the system. It is denoted by the [**MAX**] request.
2. How much each process is currently holding each resource in a system. It is denoted by the [**ALLOCATED**] resource.
3. It represents the number of each resource currently available in the system. It is denoted by the [**AVAILABLE**] resource.

Following are the important data structures terms applied in the banker's algorithm as follows:

Suppose n is the number of processes, and m is the number of each type of resource used in a computer system.

1. **Available**: It is an array of length 'm' that defines each type of resource available in the system. When Available[j] = K, means that 'K' instances of Resources type R[j] are available in the system.
2. **Max:** It is a [n x m] matrix that indicates each process P[i] can store the maximum number of resources R[j] (each type) in a system.
3. **Allocation:** It is a matrix of m x n orders that indicates the type of resources currently allocated to each process in the system. When Allocation [i, j] = K, it means that process P[i] is currently allocated K instances of Resources type R[j] in the system.
4. **Need:** It is an M x N matrix sequence representing the number of remaining resources for each process. When the Need[i] [j] = k, then process P[i] may require K more instances of resources type Rj to complete the assigned work. Nedd[i][j] = Max[i][j] - Allocation[i][j].
5. **Finish**: It is the vector of the order **m**. It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

The Banker's Algorithm is the combination of the safety algorithm and the resource request algorithm to control the processes and avoid deadlock in a system:

Safety Algorithm

It is a safety algorithm used to check whether or not a system is in a safe state or follows the safe sequence in a banker's algorithm:

1. There are two vectors **Wok** and **Finish** of length m and n in a safety algorithm.

Initialize: Work = Available
Finish[i] = false; for I = 0, 1, 2, 3, 4... n - 1.

2. Check the availability status for each type of resources [i], such as:

Need[i] <= Work
Finish[i] == false
If the i does not exist, go to step 4.

3. Work = Work +Allocation(i) // to get new resource allocation

Finish[i] = true

Go to step 2 to check the status of resource availability for the next process.

4. If Finish[i] == true; it means that the system is safe for all processes.

Resource Request Algorithm

A resource request algorithm checks how a system will behave when a process makes each type of resource request in a system as a request matrix.

Let create a resource request array R[i] for each process P[i]. If the Resource $Request_i$ [j] equal to 'K', which means the process P[i] requires 'k' instances of Resources type R[j] in the system.

1. When the number of **requested resources** of each type is less than the **Need** resources, go to step 2 and if the condition fails, which means that the process P[i] exceeds its maximum claim for the resource. As the expression suggests:

If Request(i) <= Need
Go to step 2;

2. And when the number of requested resources of each type is less than the available resource for each process, go to step (3). As the expression suggests:

If Request(i) <= Available
Else Process P[i] must wait for the resource since it is not available for use.

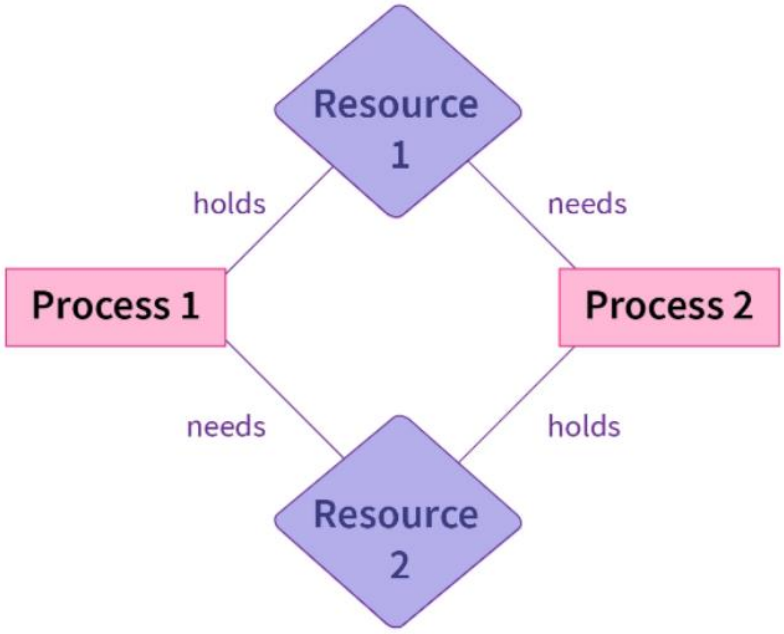
3. When the requested resource is allocated to the process by changing state:

Available = Available - Request
Allocation(i) = Allocation(i) + Request (i)
$Need_i = Need_i - Request_i$

When the resource allocation state is safe, its resources are allocated to the process P(i). And if the new state is unsafe, the Process P (i) has to wait for each type of Request R(i) and restore the old resource-allocation state.

Deadlock: A deadlock in OS is a situation in which more than one process is blocked because it is holding a resource and requires some resource that is acquired by some other process. The four necessary conditions for a deadlock situation to occur are mutual exclusion, hold and wait, no pre-emption and

circular set. We can prevent a deadlock by preventing any one of these conditions. There are different ways to detect and recover a system from deadlock. What is Deadlock in OS? All the processes in a system require some resources such as central processing unit (CPU), file storage, input/output devices, etc to execute it. Once the execution is finished, the process releases the resource it was holding. However, when many processes run on a system, they also compete for these resources they require for execution. This may arise a deadlock situation. A deadlock is a situation in which more than one process is blocked because it is holding a resource and requires some resource that is acquired by some other process. Therefore, none of the processes gets executed. Necessary Conditions for Deadlock The four necessary conditions for a deadlock to arise are as follows. • Mutual Exclusion: Only one process can use a resource at any given time i.e., the resources are non-sharable. • Hold and wait: A process is holding at least one resource at a time and is waiting to acquire other resources held by some other process. • No pre-emption: The resource can be released by a process voluntarily i.e. after execution of the process. • Circular Wait: A set of processes are waiting for each other in a circular fashion. For example, let us say there are a set of processes {P0, P1, P2, P3} such that P0 depends on P1 , P1 depends on P2, P2 depends on P3 and P3 depends on P0. This creates a circular relation between all these processes and they must wait forever to be executed.



SCALER Topics

Deadlock Avoidance in OS:

• Deadlock avoidance is a technique used in operating systems to prevent deadlocks from occurring. A deadlock is a situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

• To avoid deadlocks, the operating system can allocate resources to processes in a way that ensures that deadlock cannot occur. This can be done by using techniques such as the Banker's algorithm, which determines whether a system is in a "safe" state where no deadlock can occur.

• Another technique for avoiding deadlock is through pre-emption, which involves forcibly removing resources from one process and allocating them to another. However, preemption can be difficult to implement and can decrease system performance.

• It is important to carefully consider the costs and benefits of different approaches to deadlock avoidance, as some methods may reduce the

likelihood of deadlock but increase the overhead required to manage resources.

• Overall, the goal of deadlock avoidance is to ensure that the operating system can allocate resources to processes in a way that prevents deadlocks from occurring, while minimizing any negative impact on system performance.

Banker's Algorithm:

• The Banker's algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems.

• It was developed by Dijkstra in 1965 and is named after the hypothetical banker who gives out loans to customers.

• The algorithm prevents deadlocks in a system where multiple processes may request resources simultaneously.

• Processes need to request and release resources in a specific order to avoid deadlocks.

• The Banker's algorithm ensures that the system is in a safe state before granting a request for a resource, which means that the system will always be able to allocate resources to each process, preventing any deadlocks.

• The algorithm maintains information about the total number of available resources and the maximum number of resources each process may request.

• The algorithm checks whether a request from a process can be granted without leading to a deadlock by simulating the allocation of the resources to all processes in the system.

• The algorithm uses four data structures to keep track of the resources: Available, Max, Allocation, and Need.

• The safety algorithm simulates the allocation of resources to all processes in the system by making a copy of the Available array and checking whether there is a sequence of processes that can finish without requesting more resources than what is available.

• If such a sequence exists, the request can be granted, and the system moves to a new state. Otherwise, the request is denied, and the system remains in its current state. Overall, the Banker's algorithm provides a way to ensure that the system is in a safe state before granting a request for resources, preventing deadlocks, and ensuring that the system is able to allocate resources to each process. Advantages of Banker's Algorithm:

• Prevents deadlocks: The algorithm prevents deadlocks in a system where multiple processes may request resources simultaneously.

• Efficient: The Banker's algorithm is efficient and can handle large systems with many processes and resources.

• Avoids starvation: The algorithm ensures that each process is eventually allocated the resources it needs, avoiding starvation.

• Provides a predictable system: The algorithm provides a predictable system where processes can request and release resources without fear of deadlock or resource starvation.

Disadvantages of Bankers Algorithm:

• Assumes fixed resource requirements.

• Requires advance knowledge of resource requirements.

• May cause resource underutilization.

• Can be complex to implement.
Problem Statement:

Write a multithreaded program for preventing race conditions and deadlock avoidance for the

banker's algorithm as follows. Several customers' request and release resources from the bank. The

banker will grant a request only if it leaves the system in a safe state. A request that leaves the

system in an unsafe state will be denied.

Code:

```
#include <stdio.h>


void main()
{
   int k = 0, a = 0, b = 0;
   int no_of_resources, process;
   // Initialize variables
   printf("Enter the number of customers : ");
   scanf("%d", &process);
   printf("Enter the number of resources : ");
   scanf("%d", &no_of_resources);
   int instance[no_of_resources];
   int availability[no_of_resources];
   int allocated[process][no_of_resources];
   int need[process][no_of_resources];
   int MAX[process][no_of_resources];
   int P[process];
   int cnt = 0, i, j;
   int op[100];
   // Input the max number of instances for each resource
```

```
for (i = 0; i < no_of_resources; i++)
{
   availability[i] = 0;
   printf("Enter the max instances of resource %d : ", i);
   scanf("%d", &instance[i]);
}
printf("\nEnter the Allocation matrix \n");
// Input the allocation matrix
for (i = 0; i < process; i++)
{
   P[i] = i;
   printf("Enter the allocation of customer %d : ", i);
   for (j = 0; j < no_of_resources; j++)
   {
      scanf("%d", &allocated[i][j]);
      availability[j] += allocated[i][j];
   }
}
printf("\nEnter the Max Need matrix \n");
// Input the maximum need matrix
for (i = 0; i < no_of_resources; i++)
{
```

```
    availability[i] = instance[i] - availability[i];
  }
  printf("\n");
  for (i = 0; i < process; i++)
  {
    printf("Enter the max need of customer %d : ", i);
    for (j = 0; j < no_of_resources; j++)
    {
      scanf("%d", &MAX[i][j]);
    }
  }
  printf("\n");
// Check if granting the request for the resources will lead to a safe state or not
A:
  a = -1;
  for (i = 0; i < process; i++)
  {
    cnt = 0;
    b = P[i];
    for (j = 0; j < no_of_resources; j++)
    {
      need[b][j] = MAX[b][j] - allocated[b][j];
```

```
        if (need[b][j] <= availability[j])
            cnt++;
    }
    if (cnt == no_of_resources)
    {
        op[k++] = P[i];
        for (j = 0; j < no_of_resources; j++)
            availability[j] += allocated[b][j];
    }
    else
        P[++a] = P[i];
}
if (a != -1)
{
    process = a + 1;
    goto A;
}
// If it does, grant the request and print the safe sequence
printf("The safe sequence is : \n");
printf("<");
for (i = 0; i < k; i++)
    printf(" P[%d] ", op[i]);
```

```
    printf(">\n");

    printf("Thus request is granted\n");

}
```

Output:

```
TERMINAL                                                    Code

PS C:\Users\aspur\OneDrive\C PROGRAMS> cd "c:\Users\aspur\OneDrive\C PROGRAMS\" ; if ($?) { gcc BankersAlgo.c -o Bankers
Algo } ; if ($?) { .\BankersAlgo }
Enter the number of customers : 5
PS C:\Users\aspur\OneDrive\C PROGRAMS> cd "c:\Users\aspur\OneDrive\C PROGRAMS\" ; if ($?) { gcc BankersAlgo.c -o Bankers
Algo } ; if ($?) { .\BankersAlgo }
Enter the number of customers : 5
Enter the max instances of resource 1 : 5
Enter the max instances of resource 2 : 7

Enter the Allocation matrix
Enter the allocation of customer 0 : 0 1 0
Enter the allocation of customer 1 : 2 0 0
Enter the allocation of customer 2 : 3 0 2
Enter the allocation of customer 3 : 2 1 1
Enter the allocation of customer 4 : 0 0 2

Enter the Max Need matrix

Enter the max need of customer 0 : 7 5 3
Enter the max need of customer 1 : 3 2 2
Enter the max need of customer 2 : 9 0 2
Enter the max need of customer 3 : 2 2 2
Enter the max need of customer 4 : 4 3 3
```

```
The safe sequence is :
< P[1]  P[3]  P[4]  P[0]  P[2] >
Thus, request is granted
PS C:\Users\aspur\OneDrive\C PROGRAMS>
```

Conclusion:

Through this experiment, we gained knowledge about the techniques used to prevent Deadlock in Operating Systems, such as RAG and Banker's Algorithm. We implemented the Banker's Algorithm and acquired an understanding of its benefits, limitations, and practical uses.