Banker's Algorithm For Deadlock Detection And Avoidance

MINOR PROJECT REPORT

By

Sanyog Dani (RA2211031010087)
Arush Sirotiya (RA2211031010092)
Nikhil Kumar (RA2211031010097)
V1 Section

Under the guidance of

Dr. Saranya G

Assistant Professor , Department of Networking and Communications

In partial fulfilment for the Course

of

21CSC202J – Operating Systems

in CSE with Specialization in IT
Networking and Communications Department

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be **University** *u/s 3 of UGC Act, 1956)*

FACULTY OF ENGINEERING AND TECHNOLOGY

SCHOOL OF NETWORK AN COMMUNICATION

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR (603203)

NOVEMBER 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this B.Tech project report titled "Banker's Algorithm For Deadlock Detection And Avoidance"
is the bonafide work of

Mr. Sanyog Dani [Reg. No.: RA2211031010087]
Mr. Arush Sirotiya [Reg. No.RA2211031010092]
Mr. Nikhil Kumar [Reg. No.: RA2211031010097]

who carried out the project work under my supervision.
Certified further, that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion for this or any other candidate.

Dr. Saranya G
Assistant Professor
NWC Department

Dr. Annapurani Panaiyappan .K
Head of The Department
NWC Department

SIGNATURE OF INTERNAL EXAMINER

SIGNATURE OF EXTERNAL EXAMINER

ACKNOWLEDGEMENT

We express our heartfelt thanks to our honorable **Vice Chancellor Dr. C. MUTHAMIZHCHELVAN**, for being the beacon in all our endeavors.

We would like to express my warmth of gratitude to our **Registrar Dr. S. Ponnusamy,** for his encouragement.

We express our profound gratitude to our **Dean (College of Engineering and Technology) Dr. T. V.Gopal,** for bringing out novelty in all executions.

We would like to express my heartfelt thanks to Chairperson, School of Computing **Dr. Revathi Venkataraman,** for imparting confidence to complete my course project

We are highly thankful to our my Course project Faculty **Dr. Saranya G (Assistant Professor , Department of Networking and Communications)** for his/her assistance, timely suggestion and guidance throughout the duration of this course project.

We extend my gratitude to our **HOD Dr. Annapurani Panaiyappan .K** (**Networking and Communication**) and my Departmental colleagues for their Support.

Finally, we thank our parents and friends near and dear ones who directly and indirectly contributed to the successful completion of our project.

TABLE OF CONTENTS

Introduction

- In operating system CPU plays more important role of executing processes and controlling flow of the computing smooth.

- Deadlocks are the main problems for operating system tackle with, whenever deadlock occurs execution of processes stops completely.

- To overcome (Solve) this deadlock situation there are two algorithms one for avoidance and one for detection. In further report we are going to see take look at how they are implemented with the help of examples.

- 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.

- Banker's algorithm is used mainly in the banking system to eliminate deadlock. It helps determine whether or not a loan is being issued.

- Notations used in banker's algorithms are :-
    1. Available
    2. Max
    3. Allocation
    4. Need

- Resource request algorithm enables you to represent the machine actions when a specific process needs a resource.

- Banker's algorithm keeps many resources that satisfy the requirement of at least one client.

- The main drawback of banker's algorithm this that it does not allow the process to adjust its Maximum need during processing.

Objective

Main Goal and Objective of this Operating System Mini Project is

- To acquire knowledge of deadlock & algorithms used to detect and avoid deadlock and how they are implemented in system and there calculations to detect and avoid deadlock with help of examples.

- Banker's Algorithm is used majorly in the banking system to avoid deadlock. It helps you to identify whether a loan will be given or not.

- This algorithm is used to test for safely simulating the allocation for determining the maximum amount available for all resources.
  It also checks for all the possible activities before determining whether allocation should be continued or not.

Flowcharts and Diagrams

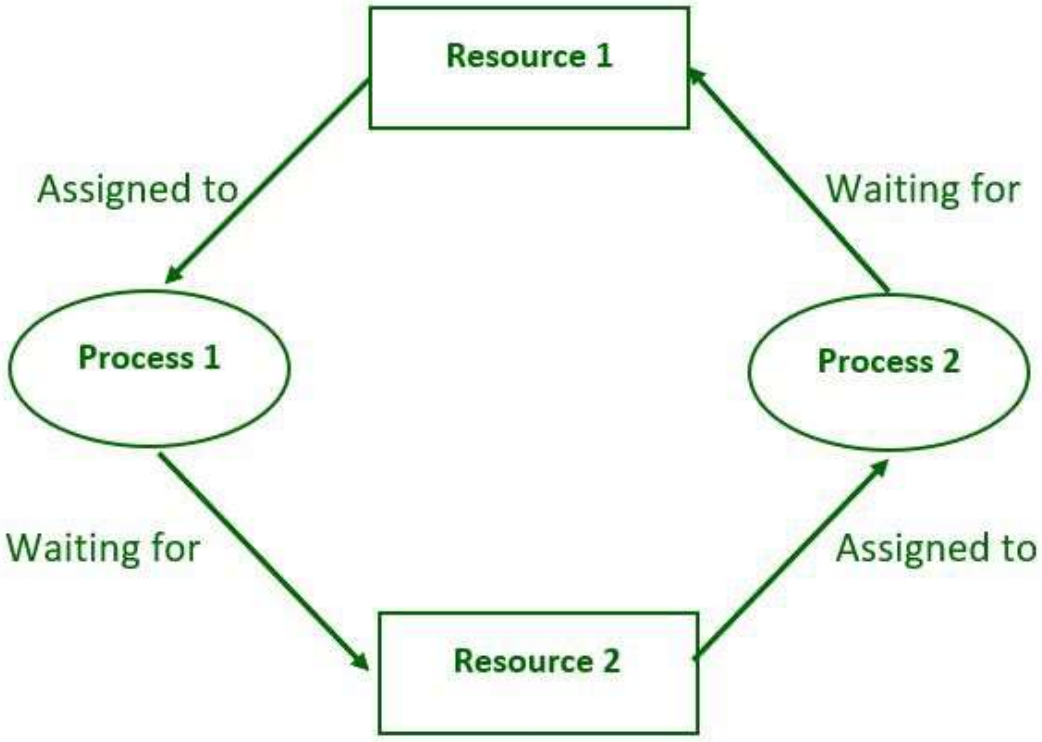
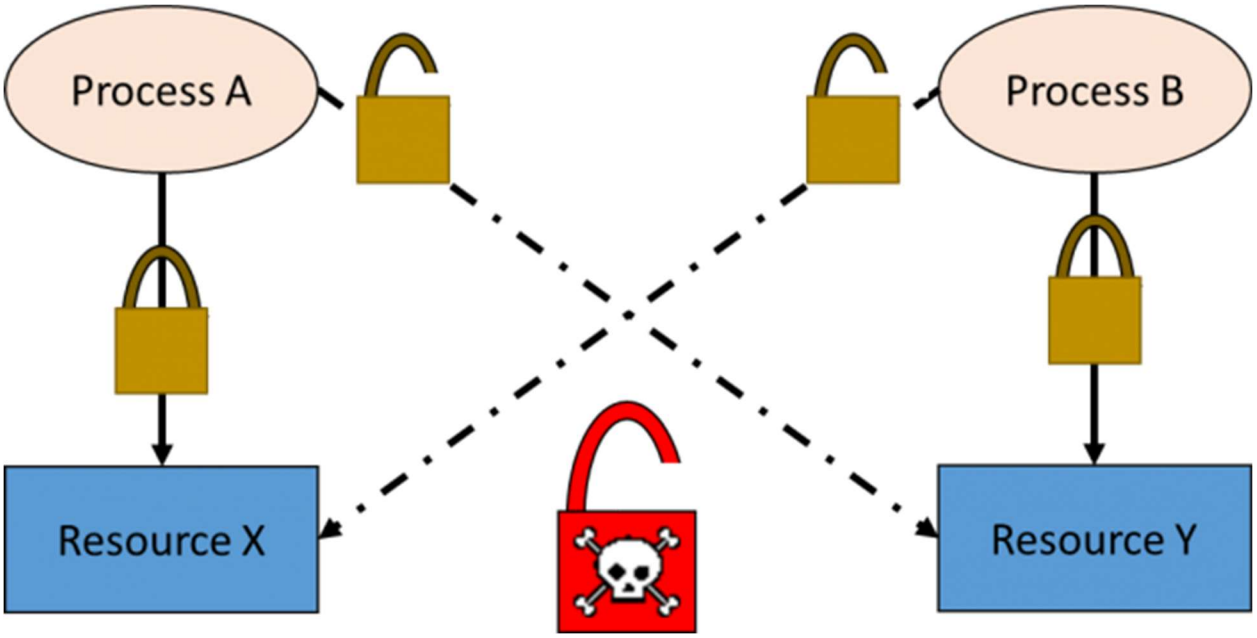
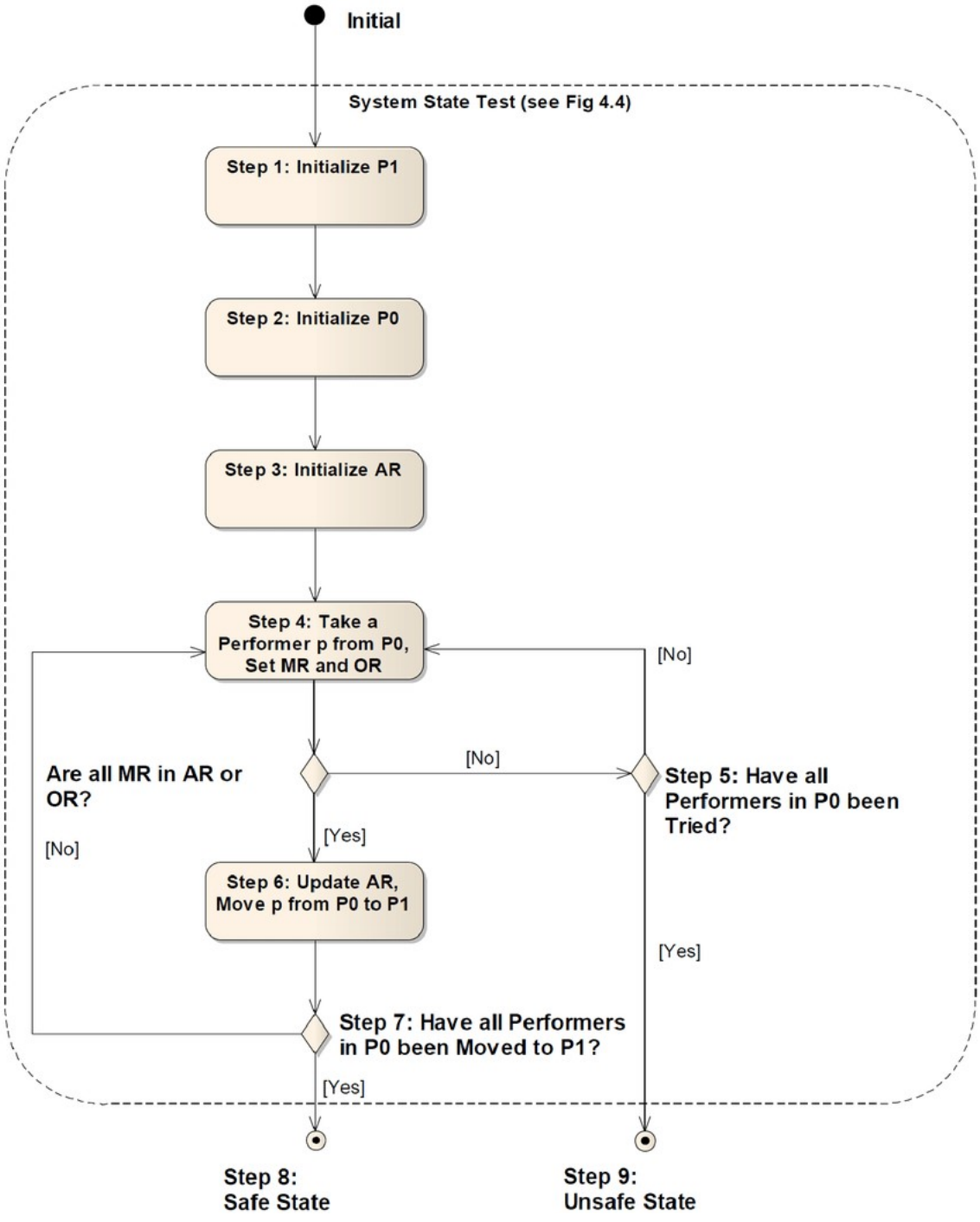


Figure: Deadlock in Operating system


Process A
Process B
Resource X
Resource Y


Initial
System State Test (see Fig 4.4)
Step 1: Initialize P1
Step 2: Initialize P0
Step 3: Initialize AR
Step 4: Take a Performer p from P0, Set MR and OR
Are all MR in AR or OR?
[No]
[Yes]
[No]
Step 5: Have all Performers in P0 been Tried?
[No]
[Yes]
Step 6: Update AR, Move p from P0 to P1
Step 7: Have all Performers in P0 been Moved to P1?
[Yes]
Step 8: Safe State
Step 9: Unsafe State

Explanation

Suppose we have **N** processes and **M** resource types in the system.

- **Available Data structure**

The first data structure which we use is **the Available Data Structure**. This data structure constructs using a one-dimensional array of size **M**(Number of available resources). If there are **K** instances of resource type $\mathbf{R_J}$, it will be denoted as:

Available[J] = K

- **Max Data structure**

Another data structure used in Banker's algorithm is **Max Data Structure**. This data structure constructs using a two-dimensional array of size **M*N**. In the system, it defines the maximum demand of each process.

If a process P_I request for **K** instances of resource type $\mathbf{R_J}$, it will be denoted as:

Max[I, J] = K

- **Allocation Data structure**

Just like **the Available** and **Max** data structure, **Allocation Data Structure** is a two-dimensional array having a size **M*N**. It defines the total number of resources allocated to a process.

If a process $\mathbf{P_I}$ is currently allocated **K** instances of resource type $\mathbf{R_J}$, it will be demoted as:

Allocation[I, J] = K

- **Need Data structure**

The **Need Data Structure** is also a two-dimensional array having a size **M*N**. It defines the total number of remaining resource which a process need.

If a process $\mathbf{P_I}$ is currently allocated **K** instances of resource type $\mathbf{R_J}$, it will be defined as:

Need [I, J] = Max[I, J] - Allocation [I, J]

In simple words, $\mathbf{(Allocation)_I}$ define the total number of resources that are currently allocated to a process $\mathbf{P_I}$, and $\mathbf{(Need)_I}$ define the total number of remaining resources for which a process $\mathbf{P_I}$ is requested for completing its task

Hardware and software Requirements

The following data structures are needed to implement the Bankers Algorithm in OS:

1. **Available Resources Array:** An array that stores the current number of available resources of each type.

2. **Maximum Need Matrix:** A matrix that stores the maximum number of resources of each type required by each process.

3. **Allocation Matrix:** A matrix that stores the number of resources of each type currently allocated to each process.

4. **Need Matrix:** A matrix that stores the remaining resources of each type required by each process (calculated as the difference between the Maximum Need and Allocation matrices).

5. **Completed Processes Set:** A set that stores the IDs of processes that have completed and released their resources.

When implementing or dealing with the Bankers algorithm in OS, three factors must be kept in mind:

1. The **[MAX]** array indicates how many resources of each type a process can request.

2. The **[ALLOCATION]** array indicates how many resources of each type a process currently holds or allocates.

3. The **[AVAILABLE]** array indicates how many resources of each type are currently available in the system.

The Bankers Algorithm is made up of two algorithms which are:

1. Safety Algorithm
2. Resource Request Algorithm

1. Safety Algorithm

- The Safety Algorithm is a part of the Bankers Algorithm in OS, which is used to determine if a system is in a safe state and if a resource request can be granted without resulting in a deadlock.

2. Resource Request Algorithm

- A resource request algorithm is a process used in operating systems to manage resource allocation. When a process requests additional resources, the resource request algorithm determines if the request can be granted without compromising the safety of the system.

- The algorithm checks if the request can be granted by ensuring that the request does not cause a process to exceed its maximum resource needs and that there are enough resources available to grant the request.

Code screenshot

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <stdbool.h>
#include <time.h>
int nResources,
 nProcesses;
int *resources;
int **allocated;
int **maxRequired;
int **need;
int *safeSeq;
int nProcessRan = 0;
pthread_mutex_t lockResources;
pthread_cond_t condition;
// get safe sequence is there is one else return false
bool getSafeSeq();
// process function
void* processCode(void* );
int main(int argc, char** argv) {
srand(time(NULL));
 printf("\nNumber of processes? ");
 scanf("%d", &nProcesses);
 printf("\nNumber of resources? ");
 scanf("%d", &nResources);
 resources = (int *)malloc(nResources * sizeof(*resources));
 printf("\nCurrently Available resources (R1 R2 ...)? ");
 for(int i=0; i<nResources; i++)
 scanf("%d", &resources[i]);
 allocated = (int **)malloc(nProcesses * sizeof(*allocated));
 for(int i=0; i<nProcesses; i++)
 allocated[i] = (int *)malloc(nResources * sizeof(**allocated));
 maxRequired = (int **)malloc(nProcesses * sizeof(*maxRequired));
 for(int i=0; i<nProcesses; i++)
 maxRequired[i] = (int *)malloc(nResources * sizeof(**maxRequired));
 // allocated
 printf("\n");
 for(int i=0; i<nProcesses; i++) {
 printf("\nResource allocated to process %d (R1 R2 ...)? ", i+1);
 for(int j=0; j<nResources; j++)
 scanf("%d", &allocated[i][j]);
 }
```

```
40      printf("\nResource allocated to process %d (R1 R2 ...)? ", i+1);
41      for(int j=0; j<nResources; j++)
42      scanf("%d", &allocated[i][j]);
43      }
44      printf("\n");
45      // maximum required resources
46      for(int i=0; i<nProcesses; i++) {
47      printf("\nMaximum resource required by process %d (R1 R2 ...)? ",
48      i+1);
49      for(int j=0; j<nResources; j++)
50      scanf("%d", &maxRequired[i][j]);
51      }
52      printf("\n");
53      // calculate need matrix
54      need = (int **)malloc(nProcesses * sizeof(*need));
55      for(int i=0; i<nProcesses; i++)
56      need[i] = (int *)malloc(nResources * sizeof(**need));
57      for(int i=0; i<nProcesses; i++)
58      for(int j=0; j<nResources; j++)
59      need[i][j] = maxRequired[i][j] - allocated[i][j];
60      // get safe sequence
61      safeSeq = (int *)malloc(nProcesses * sizeof(*safeSeq));
62      for(int i=0; i<nProcesses; i++) safeSeq[i] = -1;
63      if(!getSafeSeq()) {
64      printf("\nUnsafe State! The processes leads the system to a unsafe
65      state.\n\n");
66      exit(-1);
67      }
68      printf("\n\nSafe Sequence Found : ");
69      for(int i=0; i<nProcesses; i++) {
70      printf("%-3d", safeSeq[i]+1);
71      }
72      printf("\nExecuting Processes...\n\n");
73      sleep(1);
74      // run threads
75      pthread_t processes[nProcesses];
76      pthread_attr_t attr;
77      pthread_attr_init(&attr);
78      int processNumber[nProcesses];
79      for(int i=0; i<nProcesses; i++) processNumber[i] = i;
80      for(int i=0; i<nProcesses; i++)
81      pthread_create(&processes[i], &attr, processCode, (void
82      *)(&processNumber[i]));
```

```
80    for(int i=0; i<nProcesses; i++)
81    pthread_create(&processes[i], &attr, processCode, (void
82   *)(&processNumber[i]));
83    for(int i=0; i<nProcesses; i++)
84    pthread_join(processes[i], NULL);
85    printf("\nAll Processes Finished\n");
86   // free resources
87    free(resources);
88    for(int i=0; i<nProcesses; i++) {
89    free(allocated[i]);
90    free(maxRequired[i]);
91   free(need[i]);
92    }
93    free(allocated);
94    free(maxRequired);
95   free(need);
96    free(safeSeq);
97   }
98   bool getSafeSeq() {
99   // get safe sequence
100   int tempRes[nResources];
101   for(int i=0; i<nResources; i++) tempRes[i] = resources[i];
102   bool finished[nProcesses];
103   for(int i=0; i<nProcesses; i++) finished[i] = false;
104   int nfinished=0;
105   while(nfinished < nProcesses) {
106   bool safe = false;
107   for(int i=0; i<nProcesses; i++) {
108   if(!finished[i]) {
109   bool possible = true;
110   for(int j=0; j<nResources; j++)
111   if(need[i][j] > tempRes[j]) {
112   possible = false;
113  break;
114   }
115   if(possible) {
116   for(int j=0; j<nResources; j++)
117   tempRes[j] += allocated[i][j];
118   safeSeq[nfinished] = i;
119   finished[i] = true;
120   ++nfinished;
121   safe = true;
122   }
```

```
119     finished[i] = true;
120     ++nfinished;
121     safe = true;
122     }
123     }
124     }
125     if(!safe) {
126     for(int k=0; k<nProcesses; k++) safeSeq[k] = -1;
127     return false; // no safe sequence found
128     }
129     }
130     return true; // safe sequence found
131   }
132   // process code
133   void* processCode(void *arg) {
134     int p = *((int *) arg);
135   // lock resources
136     pthread_mutex_lock(&lockResources);
137     // condition check
138     while(p != safeSeq[nProcessRan])
139     pthread_cond_wait(&condition, &lockResources);
140   // process
141     printf("\n--> Process %d", p+1);
142     printf("\n\tAllocated : ");
143     for(int i=0; i<nResources; i++)
144     printf("%3d", allocated[p][i]);
145     printf("\n\tNeeded : ");
146     for(int i=0; i<nResources; i++)
147     printf("%3d", need[p][i]);
148     printf("\n\tAvailable : ");
149     for(int i=0; i<nResources; i++)
150     printf("%3d", resources[i]);
151     printf("\n"); sleep(1);
152     printf("\tResource Allocated!");
153     printf("\n"); sleep(1);
154     printf("\tProcess Code Running...");
155     printf("\n"); sleep(rand()%3 + 2); // process code
156     printf("\tProcess Code Completed...");
157     printf("\n"); sleep(1);
158     printf("\tProcess Releasing Resource...");
159     printf("\n"); sleep(1);
160     printf("\tResource Released!");
161   for(int i=0; i<nResources; i++)
```

```
131  }
132  // process code
133  void* processCode(void *arg) {
134   int p = *((int *) arg);
135  // lock resources
136   pthread_mutex_lock(&lockResources);
137   // condition check
138   while(p != safeSeq[nProcessRan])
139   pthread_cond_wait(&condition, &lockResources);
140  // process
141   printf("\n--> Process %d", p+1);
142   printf("\n\tAllocated : ");
143   for(int i=0; i<nResources; i++)
144   printf("%3d", allocated[p][i]);
145   printf("\n\tNeeded : ");
146   for(int i=0; i<nResources; i++)
147   printf("%3d", need[p][i]);
148   printf("\n\tAvailable : ");
149   for(int i=0; i<nResources; i++)
150   printf("%3d", resources[i]);
151   printf("\n"); sleep(1);
152   printf("\tResource Allocated!");
153   printf("\n"); sleep(1);
154   printf("\tProcess Code Running...");
155   printf("\n"); sleep(rand()%3 + 2); // process code
156   printf("\tProcess Code Completed...");
157   printf("\n"); sleep(1);
158   printf("\tProcess Releasing Resource...");
159   printf("\n"); sleep(1);
160   printf("\tResource Released!");
161  for(int i=0; i<nResources; i++)
162   resources[i] += allocated[p][i];
163   printf("\n\tNow Available : ");
164   for(int i=0; i<nResources; i++)
165   printf("%3d", resources[i]);
166   printf("\n\n");
167   sleep(1);
168  // condition broadcast
169   nProcessRan++;
170   pthread_cond_broadcast(&condition);
171   pthread_mutex_unlock(&lockResources);
172  pthread_exit(NULL);
173  }
```

Algorithm :

Step 1: When a process requests for a resource, the OS allocates it on a trial basis.

Step 2: After trial allocation, the OS updates all the matrices and vectors. This updating can be done by the OS in a separate work area in the memory.

Step 3: It compares F vector with each row of matrix B on a vector to vector basis.

Step 4: If F is smaller than each of the row in Matrix B i.e. even if all free resources are allocated to any process in Matrix B and not a single process can completes its task then OS concludes that the system is in unstable state.

Step 5: If F is greater than any row for a process in Matrix B the OS allocates all required resources for that process on a trial basis. It assumes that after completion of process, it will release all the recourses allocated to it. These resources can be added to the free vector.

Step 6: After execution of a process, it removes the row indicating executed process from both matrices.

Step 7: This algorithm will repeat the procedure step 3 for each process from the matrices and finds that all processes can complete execution without entering unsafe state.

For each request for any resource by a process OS goes through all these trials of imaginary allocation and updation. After this if the system remains in the safe state, and then changes can be made in actual matrices.

Results screenshot

```
PS C:\Users\SAN>  & 'c:\Users\SAN\.vscode\extensions\ms-vscode.
ne-Error-vfvmz5vf.u0q' '--pid=Microsoft-MIEngine-Pid-03nahrvh.l

Number of processes? 2

Number of resources? 2

Currently Available resources (R1 R2 ...)? 1
2


Resource allocated to process 1 (R1 R2 ...)? 1
2

Resource allocated to process 2 (R1 R2 ...)? 3
4


Maximum resource required by process 1 (R1 R2 ...)? 1
2

Maximum resource required by process 2 (R1 R2 ...)? 3
4



Safe Sequence Found : 1  2
Executing Processes...


--> Process 1
        Allocated :   1  2
        Needed :   0  0
        Available :   1  2
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available :   2  4


--> Process 2
        Allocated :   3  4
        Needed :   0  0
        Available :   2  4
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available :   5  8


All Processes Finished
PS C:\Users\SAN>
```

```
PS C:\Users\SAN>  & 'c:\Users\SAN\.vscode\extensions\ms-vscode.cpp
rror-rtmvdtgz.k24' '--pid=Microsoft-MIEngine-Pid-bdh0vrda.h4a' '--

Number of processes? 2

Number of resources? 2

Currently Available resources (R1 R2 ...)? 1
2


Resource allocated to process 1 (R1 R2 ...)? 3
4

Resource allocated to process 2 (R1 R2 ...)? 5
6


Maximum resource required by process 1 (R1 R2 ...)? 7
8

Maximum resource required by process 2 (R1 R2 ...)? 9
9


Unsafe State! The processes leads the system to a unsafe state.

PS C:\Users\SAN>
```

```
PS C:\Users\SAN>  & 'c:\Users\SAN\.vscode\extensions\ms-
rror-kvecvu1w.yfn' '--pid=Microsoft-MIEngine-Pid-qipckry

Number of processes? 1

Number of resources? 1

Currently Available resources (R1 R2 ...)? 3


Resource allocated to process 1 (R1 R2 ...)? 3


Maximum resource required by process 1 (R1 R2 ...)? 5



Safe Sequence Found : 1
Executing Processes...


--> Process 1
        Allocated :   3
        Needed :   2
        Available :   3
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available :   6


All Processes Finished
PS C:\Users\SAN>
```

Explanation

We will learn this with an example along with proper explanation of the whole Bnakers algorithm for deadlock avaoidance

Let us consider the following snapshot for understanding the banker's algorithm:

Process	Allocated			Maximum			Available		
	A	B	C	A	B	C	A	B	C
P0	1	1	2	4	3	3	2	1	0
P1	2	1	2	3	2	2			
P2	4	0	1	9	0	2			
P3	0	2	0	7	5	3			
P4	1	1	2	1	1	2			

The Content of the need matrix can be calculated by using the formula given below:

Need = Max – Allocation

Process	Need		
	A	B	C
P0	3	2	1
P1	1	1	0
P2	5	0	1
P3	7	3	3
P4	0	0	0

Let us now check for the safe state following the flowchart and checking all the conditions.

Safe sequence:

1. For process P0,
Need = (3, 2, 1) and
Available = (2, 1, 0)
Need <=Available = False
So, the system will move to the next process.

2. For Process P1,
Need = (1, 1, 0)
Available = (2, 1, 0)
Need <= Available = True
Request of P1 is granted.
Available = Available +Allocation
= (2, 1, 0) + (2, 1, 2)
= (4, 2, 2) (New Available)

3. For Process P2,
Need = (5, 0, 1)
Available = (4, 2, 2)
Need <=Available = False
So, the system will move to the next process

4. For Process P3,
Need = (7, 3, 3)
Available = (4, 2, 2)
Need <=Available = False
So, the system will move to the next process.

5. For Process P4,
Need = (0, 0, 0)
Available = (4, 2, 2)
Need <= Available = True
Request of P4 is granted.
Available = Available + Allocation
= (4, 2, 2) + (1, 1, 2)
= (5, 3, 4) now, (New Available)

6. Now again check for Process P2,
Need = (5, 0, 1)
Available = (5, 3, 4)
Need <= Available = True
Request of P2 is granted.
Available = Available + Allocation
= (5, 3, 4) + (4, 0, 1)
= (9, 3, 5) now, (New Available)

7. Now again check for Process P3,
Need = (7, 3, 3)
Available = (9, 3, 5)
Need <=Available = True
The request for P3 is granted.
Available = Available +Allocation
= (9, 3, 5) + (0, 2, 0) = (9, 5, 5)

8. Now again check for Process P0,
= Need (3, 2, 1)
= Available (9, 5, 5)
Need <= Available = True
So, the request will be granted to P0.

Safe sequence: < P1, P4, P2, P3, P0>

The system allocates all the needed resources to each process.

So, we can say that the system is in a safe state. The total amount of resources will be calculated by the following formula:

The total amount of resources= sum of columns of allocation + Available
= [8 5 7] + [2 1 0] = [10 6 7]

Conclusion

- We studied about that how to apply Banker's Algorithm to avoid deadlock and allocate resources safely to each process in the computer system.

- Through this algorithm we learnt how it is used in real world such as majorly in the banking system to avoid deadlock.

- It helps you to identify whether a loan will be given or not.

- This algorithm is used to test for safely simulating the allocation for determining the maximum amount available for all resources.

The application of this mini project in real world is that

- Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not.

- The banker algorithm is applied to the college scheduling system, which mainly allows the system to judge whether the existing classroom resources can meet the needs of the students in class, and find a reasonable solution, which is not a security sequence, and does not fall into a deadlock.

References

- **https://www.geeksforgeeks.org/**
- **https://www.javatpoint.com/**
- **https://www.wikipedia.org/**
- **https://www.youtube.com/**