**Contents**

[**Introduction** 3](#_Toc55767292)

[**Questions** 3](#_Toc55767293)

[Q1. How each and every function in your C programs operates? 3](#_Toc55767294)

[The Directory Structure 3](#_Toc55767295)

[Function Descriptions 3](#_Toc55767296)

[Outputs: 6](#_Toc55767297)

[Q2. Suppose no process is holding any resource at the moment. Could we allocate the required resources to process A, D, E, and F simultaneously (Shown in Table 1)? Why? 9](#_Toc55767298)

[Q3. 10](#_Toc55767299)

[3.1. Is it feasible to have x=200 and y=6? If yes will the system be in a deadlock status? 10](#_Toc55767300)

[3.2. Is it feasible to have x=300 and y=5? If yes will the system be in a deadlock status? 10](#_Toc55767301)

[Q4. Still, starting from the resource occupation in table 2 suggest a sequence of the resource allocation such that no deadlock occurs and all processes can be satisfied and terminated or provide justification if there is no such sequence of resource allocation. (Using the C code) 12](#_Toc55767302)

[Running the Code: 12](#_Toc55767303)

[Reference 15](#_Toc55767304)

# **Introduction**

In this report we will be discussing about the banker’s algorithm. The banker’s algorithm was designed for the purpose of resource allocation in the operating system. The major purpose of this algorithm is to declare whether a resource allocation state is a safe state or not. By safe sate we mean whether the resource allocation will lead to a deadlock scenario or not. This algorithm basically works by simulating all the possible allocation scenario and finding a way to allocates safely by avoiding deadlock.

# **Questions**

# Q1. How each and every function in your C programs operates?

## The Directory Structure

The Code directory contains four directories:

* Include – this directory contains the header file
* Src – this directory contains all the source codes necessary for the code to run.
* Client – this directory contains the C file which has the main method.
* Buid – this directory contains the output of the makefile and the executable file (app.exe)

Along with that an important file CMakeLists.txt is contained tin the code directory which is basically input to the CMake command in the Linux to produce the makefile for the code.

## Function Descriptions

* **Accept Function**
  + Prototype : void accept(int A[][10],int N[][10],int M[10][10],int W[1][10],int \*n,int \*m)
  + Time Complexity : O(m\*n)
  + Description:

This function takes in six parameters as an input and return nothing to the calling method. At first the function asks the user to input the number of processes and stores the value in the variable n, then it asks the use to enter the value of the resources, maximum needed allocation matrix for each of the processes, currently allocated matrix for each of the process and finally the available instances of the resources.

The this function calculates the need matrix by code snippet given below:

for(i=0;i<\*n;i++)

for(j=0;j<\*m;j++)

N[i][j]=M[i][j]-A[i][j];

And at last prints the allocation matrix, maximum requirement and the need matrix to given a better user interaction.

* + Parameter description
    - A: Allocation matrix
    - N: Need matrix
    - M: Maximum allocation matrix
    - W: Available resource
    - n: total number of process
    - m: total number of resource
* **Safety function**
  + Prototype: int safety (int A[][10],int N[][10],int AV[1][10],int n,int m,int a[])
  + Time Complexity: O(n\*n\*m)
  + Description

We are describing a flag array and initializing all the element with zero then we are creating a local array to store the presently available resource counts for array AV to W.

Now for each of the process we are running loops to check whether the currently available resources are more than any all the resources needed by the particular process, if this is the case we are allocating the resources to the process and setting the flag value to 1, while allocating the resources we actually add the allocated resources to the available resources. Like this we are try to allocate all the process if we are successful, we return 1 else we will return 0.

The code snippet give below:

for(k=0;k<n;k++){

for(i=0;i<n;i++){

if(F[i] == 0){

flag=0;

for(j=0;j<m;j++){

if(N[i][j] > W[0][j])

flag=1;

}

if(flag == 0 && F[i] == 0){

for(j=0;j<m;j++)

W[0][j]+=A[i][j];

F[i]=1;

pflag++;

a[x++]=i;

}

}

}

if(pflag == n)

return 1;

}

* **banker function**
  + Prototype: int banker(int A[][10],int N[][10],int W[1][10],int n,int m)
  + Time Complexity: O((n^2)\*m)
  + This is a straight forward function in which we are taking the following input as parameters:
    - A: Allocation matrix
    - N: Need matrix
    - W: Available resource
    - n: total number of process
    - m: total number of resource

we are calling the safety function from this function, if the function returns 0 then we can declare that deadlock has occurred and when the function return 0 we are printing the sequence of the process which can be completed or the safety sequence as it is called.

* **Res\_request function**
  + Prototype: void res\_request(int A[10][10],int N[10][10],int AV[10][10],int pid,int m)
  + Time complexity: O(m)
  + This function takes in the process number for which the additional resource allocation request is taken into account then takes in the number of instances of each resource, after that there are some validation checks like if the number of requested resources exceeds from the available resources,

for(i=0;i<m;i++)

if(reqmat[0][i] > AV[0][i]){

printf("\n Resources unavailable.\n");

exit(0);

}

Or the process number is not available

for(i=0;i<m;i++)

if(reqmat[0][i] > N[pid][i]){

printf("\n Error encountered.\n");

exit(0);

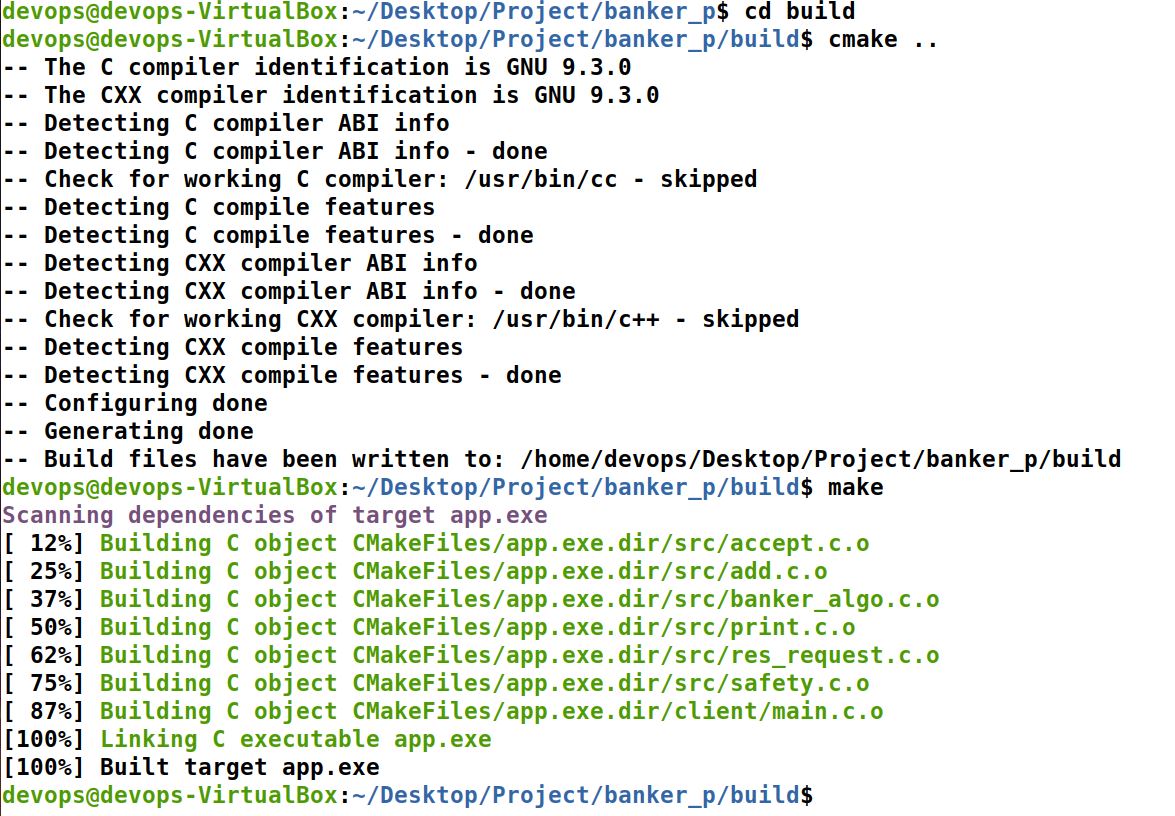
}

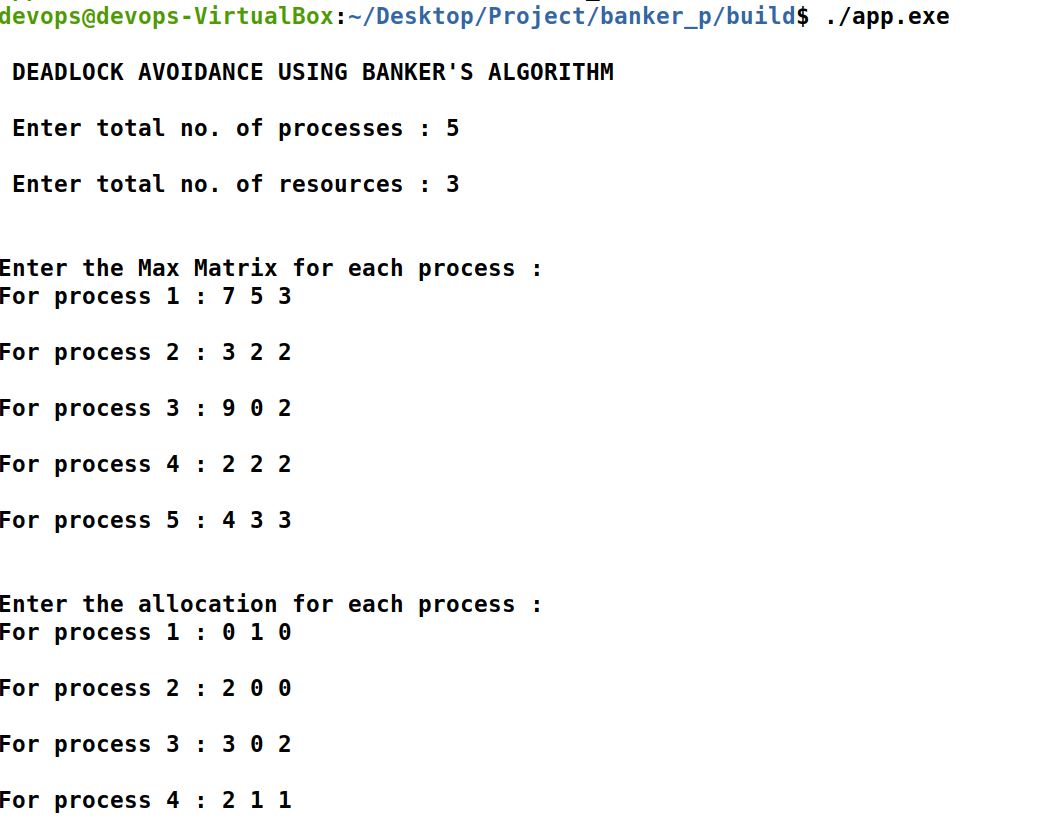
* **Print function**
  + Prototype: void print(int x[][10],int n,int m)
  + Time Complexity: O(n\*m)
  + This function takes in an array and it number of rows and columns and prints all the elements of the array.
* **main function**

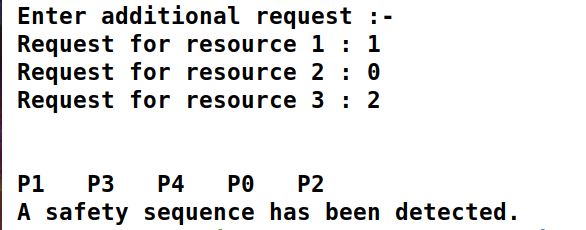
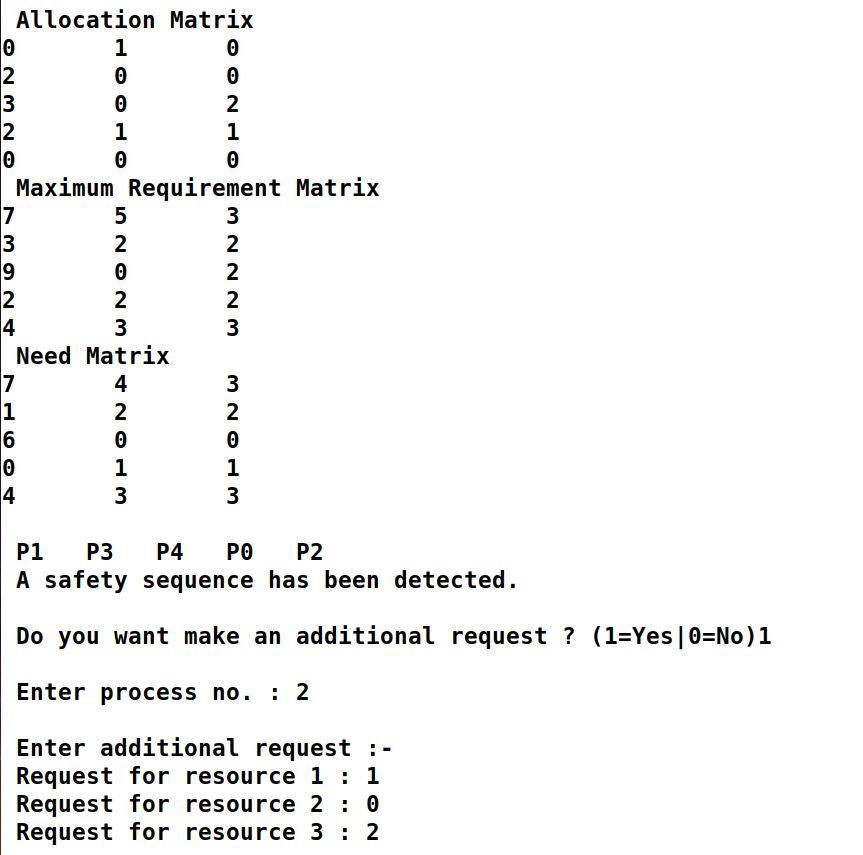
The main functions calls the accept function first then banker algorithm function and after that gives the user choice to further proceed with another resource request or exits without that.

Lastly the CMakeLists.txt file is been provided to build the program using make file.

## Outputs:







# Q2. Suppose no process is holding any resource at the moment. Could we allocate the required resources to process A, D, E, and F simultaneously (Shown in Table 1)? Why?

Solution:

Given Table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | alpha | beta | gamma | delta |
| A | 780 | 9 | 3 | 1 |
| B | 410 | 3 | 4 | 2 |
| C | 296 | 5 | 2 | 0 |
| D | 480 | 4 | 1 | 0 |
| E | 245 | 5 | 0 | 1 |
| F | 163 | 2 | 2 | 2 |

No, we cannot simultaneously allocate the resources to the processes A, D, E, and F because if we do simultaneously allocate the we will be needing

* alpha: 780+245+163+480 = 1668 instances
* beta: 9+5+2+4 = 20 instances
* gamma: 3+0+2+1 = 6 instances
* delta: 1+1+1+0 = 3 instances

while we have maximum resources as:

* alpha: 2048 instances
* beta: 16 instances
* gamma: 8 instances
* delta: 4 instances

So we can clearly see from the above needed resources and available resources that we do not have enough gamma to fulfill the requirement of the processes.

# Q3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | alpha | beta | gamma | delta |
| A | 475 | 3 | 2 | 1 |
| B | 405 | 1 | 1 | 0 |
| C | 132 | 3 | 0 | 0 |
| D | 217 | 2 | 0 | 0 |
| E | 56 | 2 | 0 | 0 |
| F | 106 | 0 | 1 | 1 |

## 3.1. Is it feasible to have x=200 and y=6? If yes will the system be in a deadlock status?

**Solution**

Total allocated resources:

* Alpha: 475+405+132+217+56+106 =1391
* Beta: 3+1+3+2+2 =11

If we try to access the for more

* Alpha: 200
* Beta: 6

Then the total needed resources will be

* Alpha: 1391+200 = 1591
* Beta: 11+6 =17

And we know that the maximum instances of beta are 16 so this request is not feasible.

## 3.2. Is it feasible to have x=300 and y=5? If yes will the system be in a deadlock status?

**Solution**

Total allocated resources:

* Alpha: 475+405+132+217+56+106 =1391
* Beta: 3+1+3+2+2 =11

If we try to access the for more

* Alpha: 300
* Beta: 5

Then the total needed resources will be

* Alpha: 1391+200 = 1691
* Beta: 11+6 =16

As all the requested resources are available then this request is feasible.

Maximum Resources needed

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | alpha | beta | gamma | delta |
| A | 780 | 9 | 3 | 1 |
| B | 410 | 3 | 4 | 2 |
| C | 296 | 5 | 2 | 0 |
| D | 480 | 4 | 1 | 0 |
| E | 245 | 5 | 0 | 1 |
| F | 163 | 2 | 2 | 2 |

Allocated matrix

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | alpha | beta | gamma | delta |
| A | 775 | 8 | 2 | 1 |
| B | 405 | 1 | 1 | 0 |
| C | 132 | 3 | 0 | 0 |
| D | 217 | 2 | 0 | 0 |
| E | 56 | 2 | 0 | 0 |
| F | 106 | 0 | 1 | 1 |

Need Matrix

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | alpha | beta | gamma | delta |
| A | 5 | 1 | 1 | 0 |
| B | 5 | 2 | 3 | 2 |
| C | 164 | 2 | 2 | 0 |
| D | 263 | 2 | 1 | 0 |
| E | 189 | 3 | 0 | 1 |
| F | 57 | 2 | 1 | 1 |

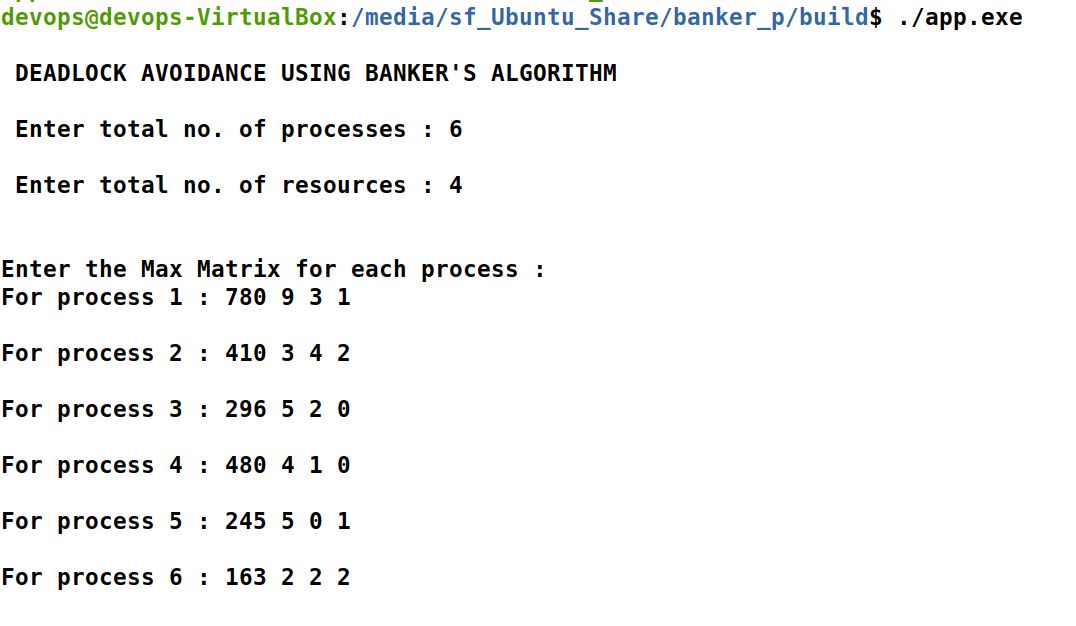
Remaining Resources

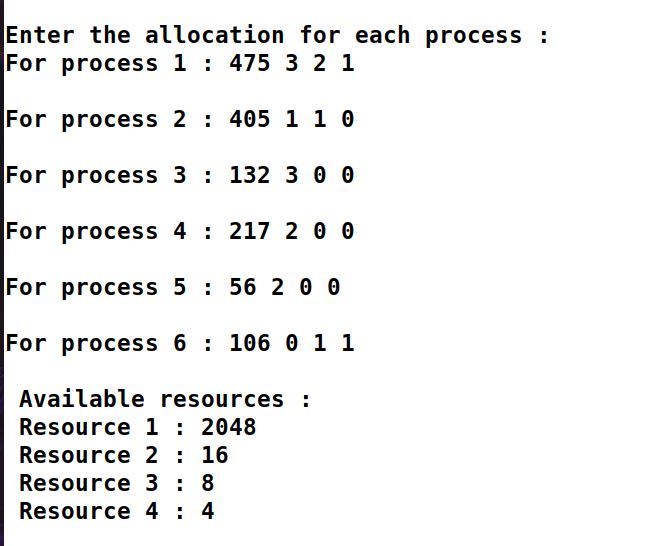
|  |  |  |  |
| --- | --- | --- | --- |
| alpha | beta | gamma | Delta |
| 358 | 0 | 4 | 2 |

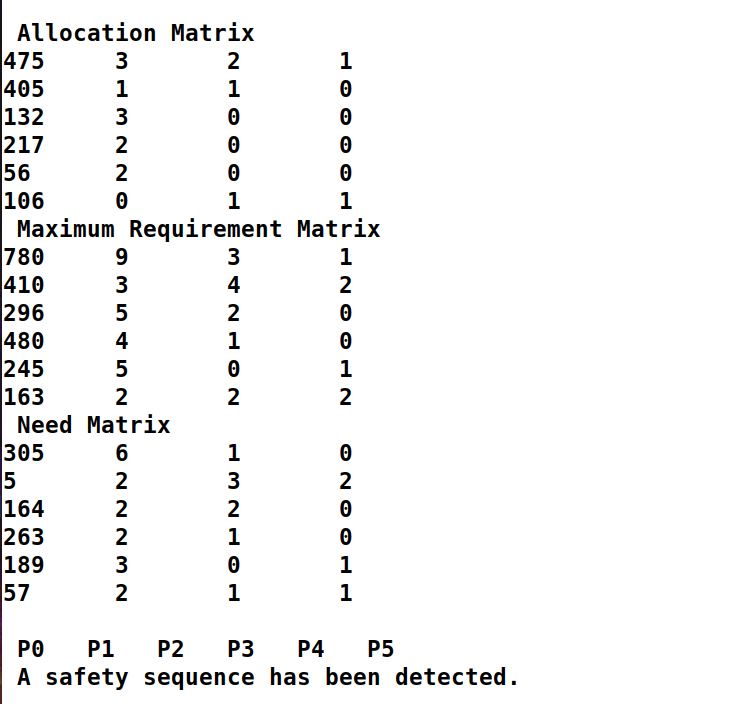
As we can clearly observe that no process can be completed with out a instance of beta hence we can conclude that this is not in safe state and deadlock occurs as all the process holds beta and waits for beta resource.

# Q4. Still, starting from the resource occupation in table 2 suggest a sequence of the resource allocation such that no deadlock occurs and all processes can be satisfied and terminated or provide justification if there is no such sequence of resource allocation. (Using the C code)

## Running the Code:







As we can see running the algorithm, we get a safety sequence as

**A->B->C->D->E->F**

# Reference

S. C. Daniels, "Real-time Conflict Resolution in Automated Guided Vehicle Scheduling", Ph.D. thesis, 1988, Dept. of Industrial Eng., Penn. State University, USA.

T. Le-Anh and M.B.M. De Koster, "A Review Of Design And Control Of Automated Guided Vehicle Systems", ERIM Report series research in management, ERS-2004-030-LIS, 2004.

J. Buffenbarger. (2012) Amake: GNU Make with automatic dependency analysis and target caching.

**.**S. Feldman, "Make: A program for maintaining computer programs," Software - Practice and Experience, vol. 9, no. 4, Apr. 1979.

R. Stallman and the GCC Developer Community. (2010) Using the GNU compiler collection. Free Software Foundation, Inc. [Online]. Available: http://www. gnu. org/software/gcc

2012) POSIX - Wikipedia, the free encyclopedia. [Online]. Available: http://en. wikipedia. org/wiki/POSIX

D. Axmark and M. Widenius. (2012) MySQL reference manual. Oracle, Inc. [Online]. Available: http://dev. mysql. com/doc