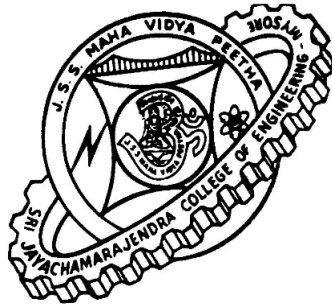


**SRI JAYACHAMARAJENDRA COLLEGE OF ENGINEERING  
MYSORE-570006**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**



**Operating System Mini-Project**  
**Implementation of Banker's Algorithm**

**Submitted by**

<b>Name</b>	<b>ROLL No</b>	<b>USN</b>
AKSHAY SHETTY	05	4JC10CS009
ASIF.M	11	4JC10CS019
PARASHIVAMURTHY	69	4JC11CS408
SRIHARSHA B.T	01	4JC09CS104

*Guidance of*

**Smt. Manimala S**

*Asst. Professor*

*Dept of CS&E, SJCE Mysore-06.*



**Affiliated to**

**Visvesvaraya Technological University  
Belgaum**

# Contents

<b>1</b>	<b>Abstract</b>	<b>1</b>
<b>2</b>	<b>Problem Definition</b>	<b>1</b>
<b>3</b>	<b>Software Requirement Specification</b>	<b>2</b>
3.1	Functional Requirements . . . . .	2
3.2	Non-functional Requirements . . . . .	2
3.3	Hardware and Software Requirements . . . . .	2
3.3.1	Hardware Requirements . . . . .	2
3.3.2	Software Requirements . . . . .	2
<b>4</b>	<b>Design Documentation</b>	<b>3</b>
4.1	Procedure Orientation Programming . . . . .	3
4.2	Context Diagram . . . . .	3
4.3	Data-Flow Diagram . . . . .	4
<b>5</b>	<b>Implementation</b>	<b>5</b>
5.1	Algorithm . . . . .	5
5.1.1	Banker's Algorithm . . . . .	6
5.2	Data Structure . . . . .	7
5.3	Error Handling . . . . .	8
<b>6</b>	<b>Result analysis</b>	<b>9</b>
6.1	Snap shots of Output . . . . .	9
6.2	Analysis on Output . . . . .	12
<b>7</b>	<b>Conclusion</b>	<b>14</b>

# 1 Abstract

The ***Banker's Algorithm*** is one of the techniques for the deadlock avoidance. The name is chosen because the algorithm could be used in the banking system to ensure that the bank never allocated less available cash in such a way that it could no longer satisfy the needs of all its customers.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. When a new process enters the system it must declare the maximum number of instances of each resource type it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise the process must wait until some other process releases enough resources.

# 2 Problem Definition

A multithreaded Program that implements the "Banker's Algorithm". Create  $n$  threads that request and releases resources from the bank. The banker will grant the request only if it leaves the system in safe state. The Program Should be written using either *pthread* or *win32* threads. It is important that access to shared data is safe from concurrent access. Such data can be safely accessed using *mutex locks*, which are available in both *threads* and *win32 API*.

**Scope:** The maximum limit on number of resources is 100, Maximum processes can be created is 100 since the maximum number of the threads can be created is 100. The each process creates a new thread. Our implementation is limited to Linux Environment provided with the pthreads library.

## 3 Software Requirement Specification

### 3.1 Functional Requirements

- Accept the number of resources and the maximum instances.
- Accept the number of processes and respective maximum resource allocation.
- Accept the current allocation for each processes.
- Banker's Algorithm to Check whether the given system is in safe state and print the safe sequences.
- A thread is created for each process requests.
- The resources are pretended to be allocated to the process and then the banker's algorithm is applied to see whether the system remains in the safe state.
- The actual allocation is done only if the system remains in the safe state.

### 3.2 Non-functional Requirements

- Program should be interactive.
- Ease of use should be flexible.
- Presentation should be good and easy to understand.
- Program is open source hence program should be written with the comments explaining the function properly.
- Documentation is done using IEEE standard with 12pt font size,oneside paper style and A4 Paper size.

### 3.3 Hardware and Software Requirements

#### 3.3.1 Hardware Requirements

A System which supports the following:

- **Processor:** Intel Pentium based System.
- **Processor speed:** 250MHz to 3.0GHz.
- **RAM:** 128MB to 8GB.

#### 3.3.2 Software Requirements

- **Operating System:** Linux Based and Windows(provided pthread is installed).
- **gcc compiler:** To compile the program and execute it.
- **PDF Reader:** To view the report.
- **LaTeX:** To prepare the documentation.

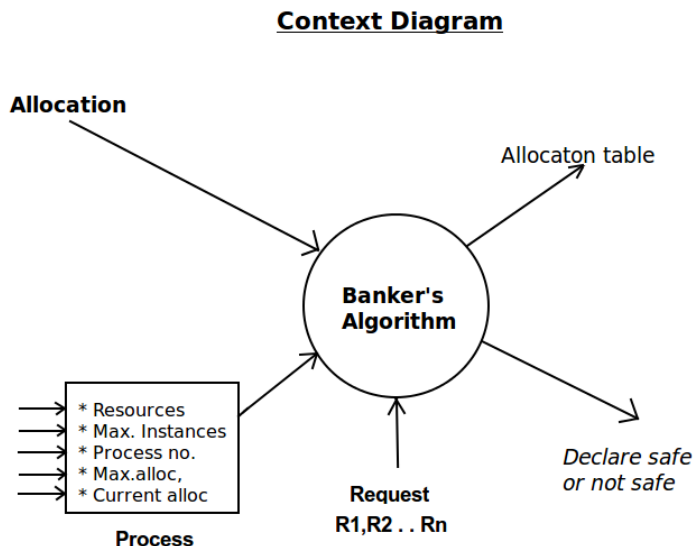
## 4 Design Documentation

### 4.1 Procedure Orientation Programming

Programming is written using C language and every functional requirement should be written using different methods/functions because to satisfy the non-functional requirement of ease of understanding the program code. The algorithm used in this implementation is explained in the next section. The main algorithm should be the banker's algorithm with two sub algorithms namely *Safety Algorithm* and *Resource Request Algorithm*.

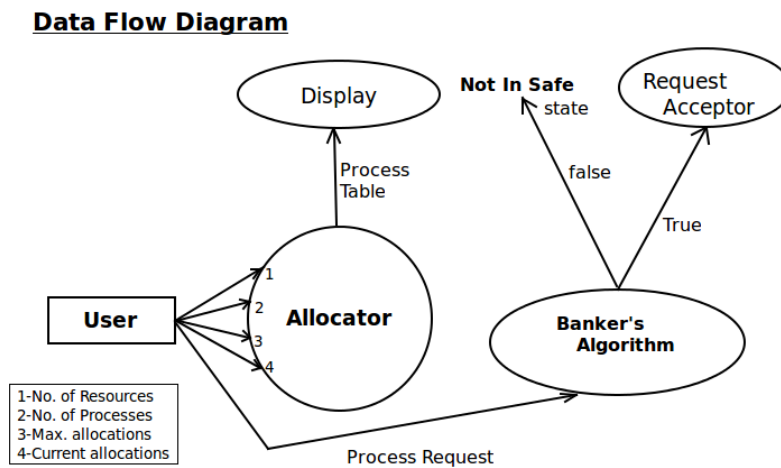
### 4.2 Context Diagram

The below figure shows the context flow of the process in the implementation of the Banker's Algorithm. Here number of resources, maximum instances of the resources, number of processes, maximum allocation of request to each process and current allocation of the resources by the process is given to the banker's algorithm along with the allocation details and requests by each process. Algorithm decides whether the system is in safe state or not. If the system is in safe state it allocates the resources to the requested process else it will ask hold the request for further process.



### 4.3 Data-Flow Diagram

The data flow diagram shown here consists of a user and various sub systems. User give the information about the resources and the processes, these information is used by allocator and it allocates the processes and give a Structured *Process Table* to the user. Further user give the request to the *Banker's algorithm* and these request are processed by *Banker's Algorithm* and decide whether the system is in safe state or not. On safe state the request is accepted else and error is displayed to the user.



## 5 Implementation

Implementation should be straight forward. Program should accept the *input*, The input accept should be error free and if user enters an input which causes an error in the system that should be specified. And the input should be processed using *Banker's Algorithm* and recursive program should be return for the resource request algorithm and data structure should be effective enough for the dynamic processes created. Display the results in a good presentation form so that user has no ambiguity in the result obtained.

As problem defined threads to be created for each processes and processes should be accepting the requests. Hence programmer should be extra cautious about Multi-thread handling and its synchronizations. proper *wait()* functions and *sleep()* function be used to gain synchronization.

### 5.1 Algorithm

The Implementation algorithm with the computation of the required intermediate inputs for the Banker's Algorithm is should be as follow:

Step 0: Start

Step 1: Read the number of Resources and its Maximum Instances  
[Read nR and MI[0..nR] ]

Step 2: Read the number of processes and maximum allocation of resources by each process on each resources  
[Read nP and MxAl[0..nP] ]

Step 3: Read the current Allocation by each processes.

check **for** error

10

**for** each processes

**if** CrAl[i] > MxAl[i]

display error

**else**

**goto** to step 4

Step 4: Allocate the Resources to processes

[Assign CrAl[0..nR] ]

Step 5: [Find Available]

20

**for** i in each resources

Available[i] = (MI[i] - Sum(CrAl[i]))

Step 6: [calculate Need[0..nP][0..nR]  
           **for** i in each process  
               Need[i] = MxAl[i]–CrAl[i]

Step 7: [Check **for** Safe State using safety algorithm]  
           Safety Algorithm() 30

Step 8: Read the request by each process  
           [Read Req[0..nR] ]

Step 9: [**for** each requests apply resource–request algorithm]  
           Resource–Request Algorithm()

Step 10: **if** safe state  
               display sequence of processes  
           **else**  
               display Error 40

Step 11: end

### 5.1.1 Banker’s Algorithm

#### Safety Algorithm

Step 1: Initialize  
           Work = Available  
           **for** i=0..nP  
               Finish[i]=false

Step 2: Find an i such that both  
           a. Finish[i]==false  
           b. Need[i]<=Work  
           **if** no such i exists go to Step 4 10

Step 3: Work=Work + Allocation  
           Finish[i]=true  
           Go to Step 2.

Step 4: If Finish[i]==true **for** all i, then the System is in a safe State



## Resource Request Algorithm

- Step 1: **if** Request[i] <= Need[i], go to Step 2.  
Otherwise, raise an Error condition,  
since the process has exceeded its maximum claim.
- Step 2: **if** Request[i] <= Available, go to Step 3.  
Otherwise, P[i] must wait,  
since the resources are not available.
- Step 3: Have the system pretend to have allocated the requested  
resources to process P[i] by modifying the state as follows: 10  
(Safety Algorithm should be Used)  
Available = Available - Request;  
Allocation[i] = Allocation[i] + Request;  
Need[i] = Need[i] - Request;

If the resulting resource allocation state is safe, The transaction is completed, And process  $P_i$  is allocated its resources. However, If the New state is unsafe, Then  $P_i$  must wait for  $Request_i$ , and the old resources-allocation state is restored.

## 5.2 Data Structure

Basically Banker's Algorithm uses the Matrix as its data structure, where number of processes is represented by the rows of matrix and number of resources as columns of the same matrix.

But in our implementation we need to create threads for each process and hence should accept the request from the respective. Hence, matrix data structure is not effective in this implementation. So we choose **Structure** as our data structure which is used in C programming. Where Array of Structure is maintained for each process. Where *structures* should have members for MAXIMUM ALLOCATIONS, CURRENT ALLOCATIONS, REQUEST OF EACH PROCESS AND NEED OF PROCESSES. All the data structure are of type integers (**int**). For intermediate outputs like Available and Maximum Instances we should use array of integers. A declared Structure with its members is given below.

```

struct Process
{
    int pid;           /*Count for No. of Process*/
    int MxAl[nR]; /*Maximum Allocation array*/
    int CrAl[nR]; /*Current Allocation array*/
    int Req[nR];  /*Request Array */
    int Need[nR]; /*Need array */
                  /* Where nR is No. of Resources */
};

```

10

A single Structures to be defined for each process which done by each threads created for every process

### 5.3 Error Handling

Durring implementation many errors can be occured and from the require-ment analysis we have a list of few errors to be handled durring implemen-tations.

#### Errors by User's Input:

- Error to be handled when User enter the Allocated resources value greater than Maximum Required resources, which leads to a negative value in Need array which can unstablize the system's calculations.  
i.e, Need=Max-Allocated;
- Number of resources ad number of Processes are constrained to some Maximum value hence the input should be under that constraints.

#### Errors while using Multi-threads:

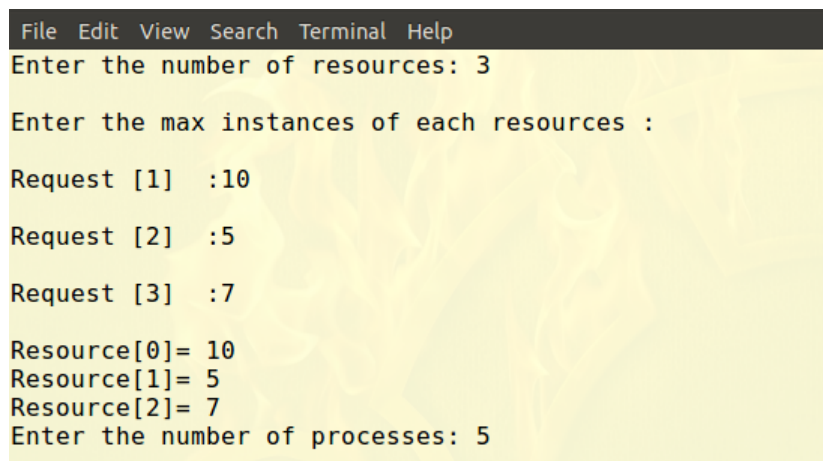
- Since the implementation is a multithreaded, the creation of the threads should be maintained properly, i.e, every thread created should be ter-minated when the task is done and every thread should join the main process after its task is ended.
- Threads requested for the mutex some times need to wait for the lock to be unlocken by other processes(threads), so threads wait should be handled well so that it should not create any dead lock effect on the system.

### Errors on Mutex:

- Since the data should be in mutex, so we could use *pthread\_mutex* or *win32\_mutex* and its methods to lock and unlock the mutex.
- Programmer should Keep in mind that the lock of the Mutex should be given only if system is in safe state or going to be in safe state.
- No part of any methods used in the implementation should keep the mutex to be locked for long period that may result in the more waiting time for other threads.

## 6 Result analysis

### 6.1 Snap shots of Output



```
File Edit View Search Terminal Help
Enter the number of resources: 3

Enter the max instances of each resources :

Request [1] :10
Request [2] :5
Request [3] :7

Resource[0]= 10
Resource[1]= 5
Resource[2]= 7
Enter the number of processes: 5
```

Figure 1: Reading Resources and its Maximum Instances

```
File Edit View Search Terminal Help
For process 1
Enter for resource 1 : 7

Enter for resource 2 : 5

Enter for resource 3 : 3

For process 2
Enter for resource 1 : 3

Enter for resource 2 : 2

Enter for resource 3 : 2

For process 3
Enter for resource 1 : 9

Enter for resource 2 : 0

Enter for resource 3 : 2

For process 4
Enter for resource 1 : 2

Enter for resource 2 : 2

Enter for resource 3 : 2

For process 5
Enter for resource 1 : 4

Enter for resource 2 : 3

Enter for resource 3 : 3
```

Figure 2: Reading Process and Max Allocation

```
File Edit View Search Terminal Help
Enter the allocation for each process:

For process 1 :
Enter for resource 1 : 0

Enter for resource 2 : 1

Enter for resource 3 : 0

For process 2 :
Enter for resource 1 : 2

Enter for resource 2 : 0

Enter for resource 3 : 0

For process 3 :
Enter for resource 1 : 3

Enter for resource 2 : 0

Enter for resource 3 : 2

For process 4 :
Enter for resource 1 : 2

Enter for resource 2 : 1

Enter for resource 3 : 1

For process 5 :
Enter for resource 1 : 0

Enter for resource 2 : 0
```

Figure 3: Reading Current Allocation by each process

The system is in safe state

The safe state is < P2 P4 P5 P1 P3 >

Process	Allocated	Maximum	Need
1	< 0 1 0 >	< 7 5 3 >	< 7 4 3 >
2	< 2 0 0 >	< 3 2 2 >	< 1 2 2 >
3	< 3 0 2 >	< 9 0 2 >	< 6 0 0 >
4	< 2 1 1 >	< 2 2 2 >	< 0 1 1 >
5	< 0 0 2 >	< 4 3 3 >	< 4 3 1 >

AVAILABLE: < 3 3 2 >

Thread created for process 1

Enter the request of the resources by the process 1: █

Figure 4: Displaying Need And Allocation With safe state sequence before any request from process

## 6.2 Analysis on Output

In fig.1 to 3 system is accepting the input such as Number of resources, Maximum instances of the resources, Number of processes and Maximum allocation and Current allocation for each processes. Here the inputs are give with no. error and hence system continue and check whether the current allocation keep system safe of not, this result is shown in fig-4. In fig-4 system is accepting request of process 1 and this request is processed and check whether allocation of this request is safe or not, and the proper result displayed with the process sequence up to that request.

Fig 5 showing the allocation of safe request by various process. Fig 6 shows that process 5 requesting an unstable resource allocation. Our program analyse this request and since this request allocation will cause unsafe state of the system that request is not granted by the program.

And Hence Banker's Algorithm avoids the dead lock that can caused by the resource request by process 5.

```
File Edit View Search Terminal Help
Enter the request of the resources by the process 2: 1 0 2

The system is in safe state and the request of process 2 is accepted
The safe sequence is < P2 P4 P5 P1 P3 >
```

Process	Allocated	Maximum	Need
1	< 0 1 0 >	< 7 5 3 >	< 7 4 3 >
2	< 3 0 2 >	< 3 2 2 >	< 0 2 0 >
3	< 3 0 2 >	< 9 0 2 >	< 6 0 0 >
4	< 2 1 1 >	< 2 2 2 >	< 0 1 1 >
5	< 0 0 2 >	< 4 3 3 >	< 4 3 1 >

```
AVAILABLE: < 2 3 0 >

End of thread for process 2

Thread created for process 3
Enter the request of the resources by the process 3: █
```

Figure 5: Safe state after accepting request from process 2

```
File Edit View Search Terminal Help

The safe sequence is < P2 P4 P5 P1 P3 >
```

Process	Allocated	Maximum	Need
1	< 0 1 0 >	< 7 5 3 >	< 7 4 3 >
2	< 3 0 2 >	< 3 2 2 >	< 0 2 0 >
3	< 3 0 2 >	< 9 0 2 >	< 6 0 0 >
4	< 2 1 1 >	< 2 2 2 >	< 0 1 1 >
5	< 0 0 2 >	< 4 3 3 >	< 4 3 1 >

```
AVAILABLE: < 2 3 0 >

End of thread for process 4

Thread created for process 5
Enter the request of the resources by the process 5: 3 3 0

The system is in unsafe state and hence the request is not accepted
End of thread for process 5
```

Figure 6: Showing Unsafe state on accepting process 5's request

## 7 Conclusion

The working and implementation of the Banker's Algorithm is shown above using figures and theorital analysis. For a given set of processes and resources, our program is creating different multithreads for each processes and accepting the resource request by the same and processing the inputs according to the Banker's Algorithm and producing the safe state sequence and Error if the system is unsafe.

### References:

#### Text Book:

Operating system Principles - 8th Edition

#### Websites:

1. [http://en.wikipedia.org/wiki/Banker/27s\\_algorithm](http://en.wikipedia.org/wiki/Banker/27s_algorithm)
2. [http://en.wikipedia.org/wiki/POSIX\\_Threads](http://en.wikipedia.org/wiki/POSIX_Threads)
3. <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>