

**COURSE CODE : CSE 316**

**SUBJECT : OPERATING SYSTEMS**

**SIMULATION BASED PROJECT**

**ASSIGNMENT QUESTION - 3**

**StudentName :** Suraj Kumar Das

**Student ID :** 11607097

**Email Address :** surajdas1818@gmail.com

**GitHub Link :** https://github.com/surajkumardas/Bankers-s-Algorithm

**Question Code:** Question 3

**Question 3 :** Write a multithreaded program that implements the banker's algorithm. Create n threads that request and release resources from the bank. The banker will grant the request only if it leaves the system in a safe state. It is important that shared data be safe from concurrent access. To ensure safe access to shared data, you can use mutex locks.

**CODE** **:**

1. #include <pthread.h>
2. #include <unistd.h>
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <fcntl.h>
6. int pmax,rmax;
7. int safe\_state = 0;
8. pthread\_t bankers[3][3];
9. int Avail[3];
10. int Max\_Resource[3][3]={{10,10,10},{10,10,10},{10,10,10}};
11. int Alloc[3][3]={{1,2,1},{1,1,1},{1,2,3}};
12. int Need[3][3];
13. int cp=0, cr=0, tp=3, tr=3;
14. pthread\_mutex\_t locking;
15. void \* locate\_function(void \* re);
16. void \* check\_function(void \* re);
17. int main()
18. {
    1. printf("\n \t\t\t\t \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Banker's Algorithm \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \t\t\t\t \n");
    2. pthread\_mutex\_init(&locking, NULL);
    3. printf("\n Enter the maximum number of Processes: \n ");
    4. scanf("%d",&pmax);
    5. printf("\n Enter the maximum number of Resources: \n");
    6. scanf("%d",&rmax);
    7. printf("\nThe Max Resources entered is: \n");
    8. for(int i=0; i<3; i++)
    9. {
       1. for(int j=0; j<3; j++)
       2. {
          1. printf("%d",Max\_Resource[i][j]);
       3. }
       4. printf("\n");
    10. }
    11. printf("\nThe Allocated Resources are: \n");
    12. for(int i=0; i<3; i++)
    13. {
        1. for(int j=0; j<3; j++)
        2. {
           1. printf("%d",Alloc[i][j]);
        3. }
        4. printf("\n");
    14. }
    15. Avail[rmax];
    16. int i,j;
    17. int res1=3, res2=2, res3=2;
    18. printf("\n Mutex Lock Initialized\n");
    19. pthread\_create(&bankers[0][0], NULL, &check\_function, (void \*)res1);
    20. pthread\_create(&bankers[1][0], NULL, &locate\_function, (void \*)res2);
    21. pthread\_create(&bankers[2][0], NULL, &locate\_function, (void \*)res3);
    22. for(i=0; i<tp; i++)
    23. {
        1. for(j=0; j<tr; j++)
        2. {
           1. pthread\_join(bankers[0][0], NULL);
           2. pthread\_join(bankers[1][0], NULL);
           3. pthread\_join(bankers[2][0], NULL);

iii.}

* 1. }
  2. return 0;

1. }
2. void \* check\_function(void \* re)
3. {
   1. int t\_no = (int) re;
   2. int x,y;
   3. printf("\n Thread Start with number: %d\n", t\_no);
   4. while(cp<pmax && cr<rmax)
   5. {
      1. pthread\_mutex\_lock(&locking);
      2. printf("\n Mutex Lock Enabled \n");
      3. printf("\n Checking Availablity List: \n");
      4. Avail[x] = Max\_Resource[x][y] - Alloc[cp++][cr++];
      5. printf("\n Availabe is: %d\n",Avail[x]);
      6. printf("check\_function:\n thread : %d,\n Available : %d.\n Signalling done and Received.\n",t\_no,Avail[y]);
      7. cr = cr+1;
      8. printf("check\_function:\n thread : %d,\n New Need Now : %d.\n",t\_no,Need[cp][cr]);
   6. }
   7. pthread\_mutex\_unlock(&locking);
   8. printf("\n Mutex Lock Released \n");
4. }
5. void \* locate\_function(void \* re)
6. {
   1. int x,y,i,j;
   2. int t\_no = (int) re;
   3. for(i=0;i<pmax;i++)
   4. {
      1. for(j=0;j<rmax;j++)
      2. {
         1. printf("\n Calculating Need for Location \n");
         2. Need[i][j] = Max\_Resource[i][j]-Alloc[i][j];
         3. printf("\n Allocation : %d,\n Need : %d\n",Alloc[i][j],Need[i][j]);
         4. if(Need[i][j] <= Avail[i])
         5. {
            1. pthread\_mutex\_lock(&locking);
            2. printf("\n Mutex Lock Enabled \n");
            3. safe\_state = 1;
            4. printf("\n Resource is allocated.\n");
            5. printf("\n After Resource is Free.The New Avail is : \n");
            6. Avail[i] = Avail[i]+Alloc[i][j];
            7. printf("%d\n",Avail[i]);
            8. pthread\_mutex\_unlock(&locking);
            9. printf("\n Mutex Lock Released \n");
         6. }
         7. else
         8. {
            1. printf("\n Unsafe State.\n Resource can't be allocated.\n");
            2. safe\_state = 0;
            3. break;
         9. }

iii.}

* + 1. pthread\_mutex\_lock(&locking);
    2. if(cp==pmax && cr==rmax)
    3. {
    4. printf("\nlocate\_function:\n thread %d,\n Need = %d.\n Limit Reached.\n",t\_no,Need[i][j]);
    5. }
    6. pthread\_mutex\_unlock(&locking);
    7. printf("\n locate\_function:\n thread %d,\n nNeed = %d.\n Locking Mutex Unlocked.\n",t\_no,Need[i][j]);
    8. sleep(1);
    9. printf("\n Going to Check Function...\n");
    10. check\_function(re);
  1. }

1. }
2. **Explain the problem in terms of operating system concept?**

**Banker’s Algorithm :** The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation of resources to processes on their request and then makes a safe-state check to test whether the request for resource should be granted or not.

**Safe State** : It is the condition of the state of system if the system can allocate all resources requested by all processes i.e. up to their stated maximums without entering a deadlock state, If a safe sequence does not exist, then the system is in an unsafe state, which may lead to deadlock.

**Unsafe State** : The state of the system where the system is in deadlock state after the allocation of resources to processes.

**DeadLock**  : In an operating system, a deadlock occurs when a process or thread enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process.

**2.**.**Write the algorithm for proposed solution of the assigned problem.**

1. Let Work and Finish be vectors of length m and n, respectively. Initialize

Work - Available. For i = 0 , 1 , . . . , n-1, if Allocation != 0, then

Finish[i] = false;

otherwise, Finish[i] = true.

2. Find an index i such that both

a. Finish[i] == false

b. Request*i* <= Work

If no such i exists, go to step 4.

3. Work = Work + Allocation

Finish[i] = true

Go to step 2.

4. If Finish[i] == false, for some i, 0 <= i< n, then the system is in a deadlocked

state. Moreover, if Finish[i] == false, then process P; is deadlocked.

**3.Calculate complexity of implemented algorithm. (Student must specify complexity of each line of code along with overall complexity)**

Solution : The Time Complexity of Banker’s Algorihtm is as follow

|  |  |
| --- | --- |
| Line Of Code | Time Complexity |
| For line 18 ( h – j ) | O(n^2) |
| For line 18 ( l – n ) | O(n^2) |
| For line 18 ( v – x ) | O(n^2) |
| For line 21 ( d - f ) | O(n) |
| For line 24 ( d – f ) | O(n^2) |
| For Whole Banker’s Algorithm  i.e. Overall Complexity | O(n\*n\*m), where n is number of processes and  m is number of resources. |

**4. Explain all the constraints given in the problem. Attach the code snippet of the implemented constraint.**

**1. Max\_Resources** **:** We need this constraint as this helps in allocating the maximum resources needed by the process and any request of the process can not be greater than the maximum resources of the process.

**2. Availability :** We need this constraint as this helps in telling the available resource of any process at current time, the availability is calculated as

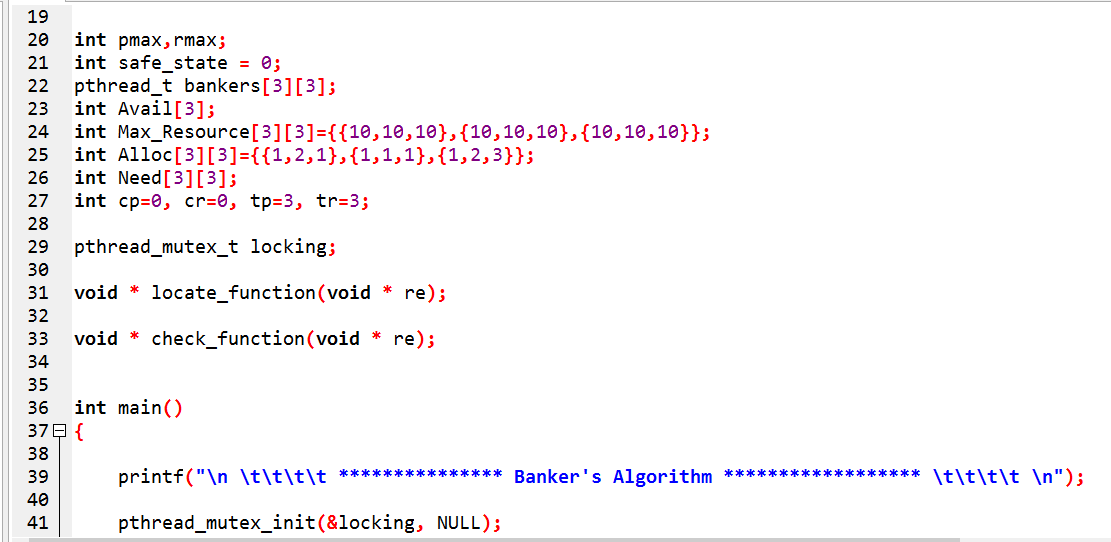
Availability [x] = Availability [x] + Allocation[x][y];

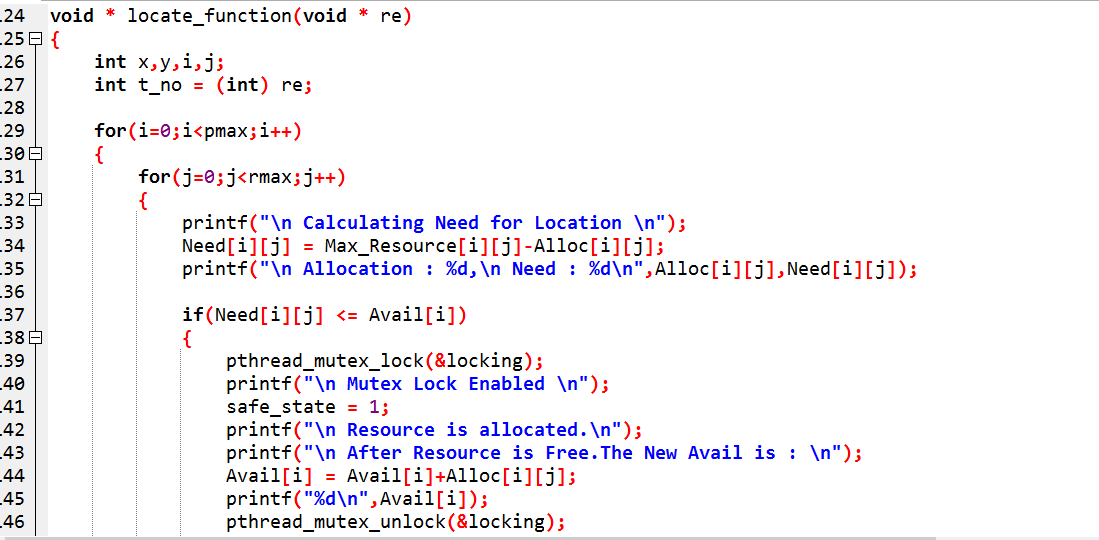
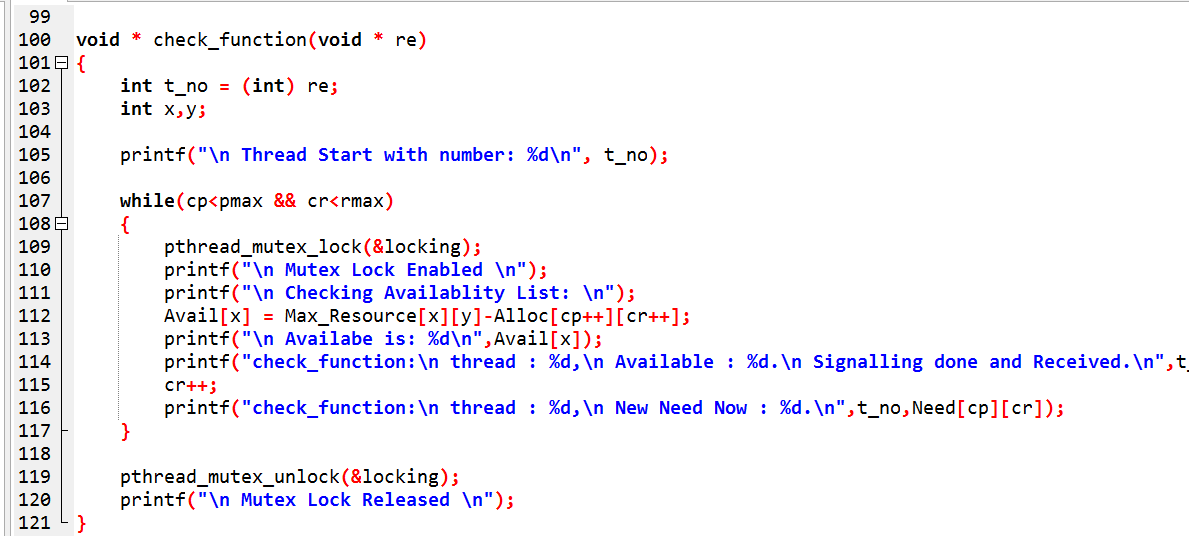
If Need of any process is greater than Availability then the resource can’t be accepted.

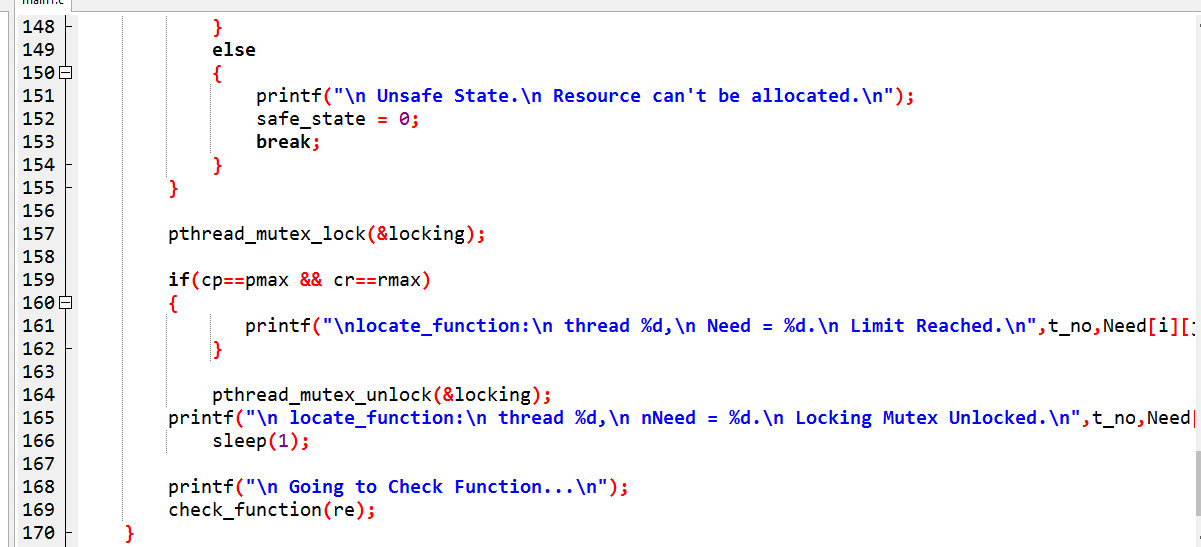
**3.Need :** We need this constraint as this helps in telling the needed resource of any process at current time, the need is calculated as

Need [x][y ] = Max\_Resources[x][y] – Allocation[x][y];

**4. Check And Locate Function :** These are the two functions of the banker’s algorithm that helps in checking the requirements and depending upon the requirement the request is granted and the state of the system is in safe state, if any request results in unsafe state of the system than that request is not granted.



****



**5. Purpose of Use Of Banker’s Algorithm**

Banker's algorithm is a deadlock avoidance algorithm. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not. It is used to get the safe sequence so that no request from the process side is granted that affects the system and lead it to unsafe state i.e. deadlock state.

Through this only those request are granted that does not leave the system in unsafe state.

**6. Boundary Condition of the code implemented**

1. No of Processes is restricted to only 3 i.e. it will not work for more than 3 processes.

Here the thread that has been created in only restricted to only 3 processes

Bankers[0][0], Bankers[1][0], Bankers[2][0]

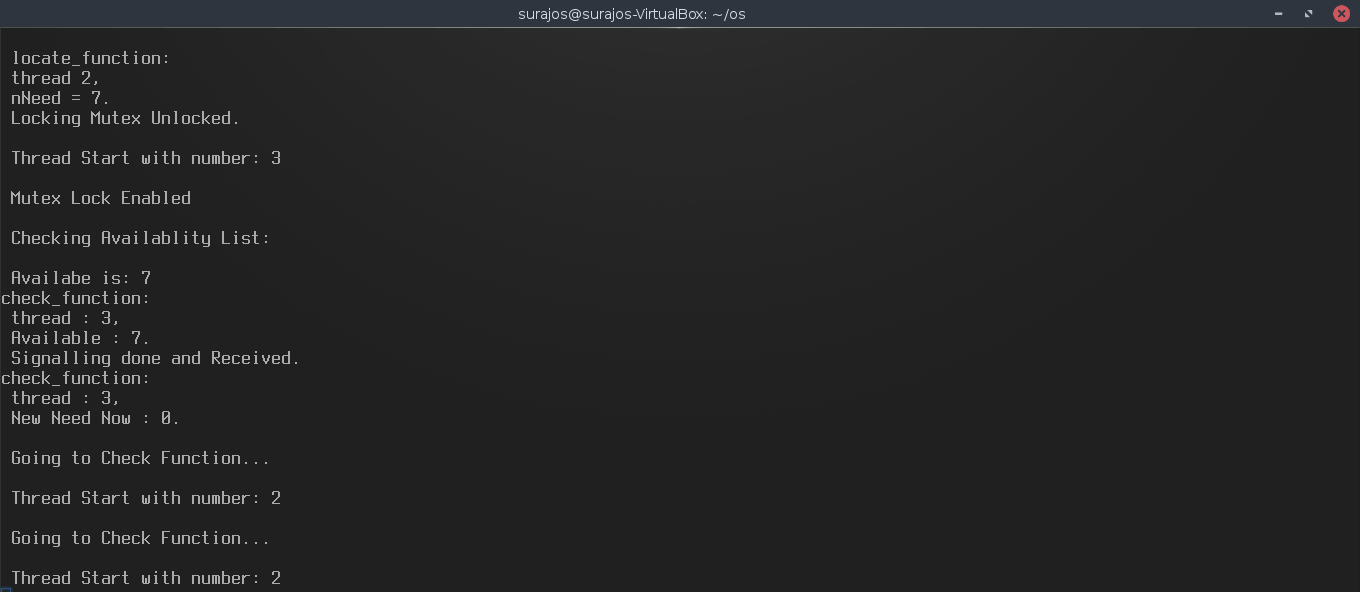
If it exceeds more than 3 process than the code will not work.

1. The maximum resource that can be granted to any processor is limited to 10. i.e. no more than 10 resource can be granted to any process. If any request for resource is greater than 10 and is the request is granted than the system in dead lock state as all other resources will be in starvation for infinite time i.e. in dead lock state.

**7. TEST CASE**

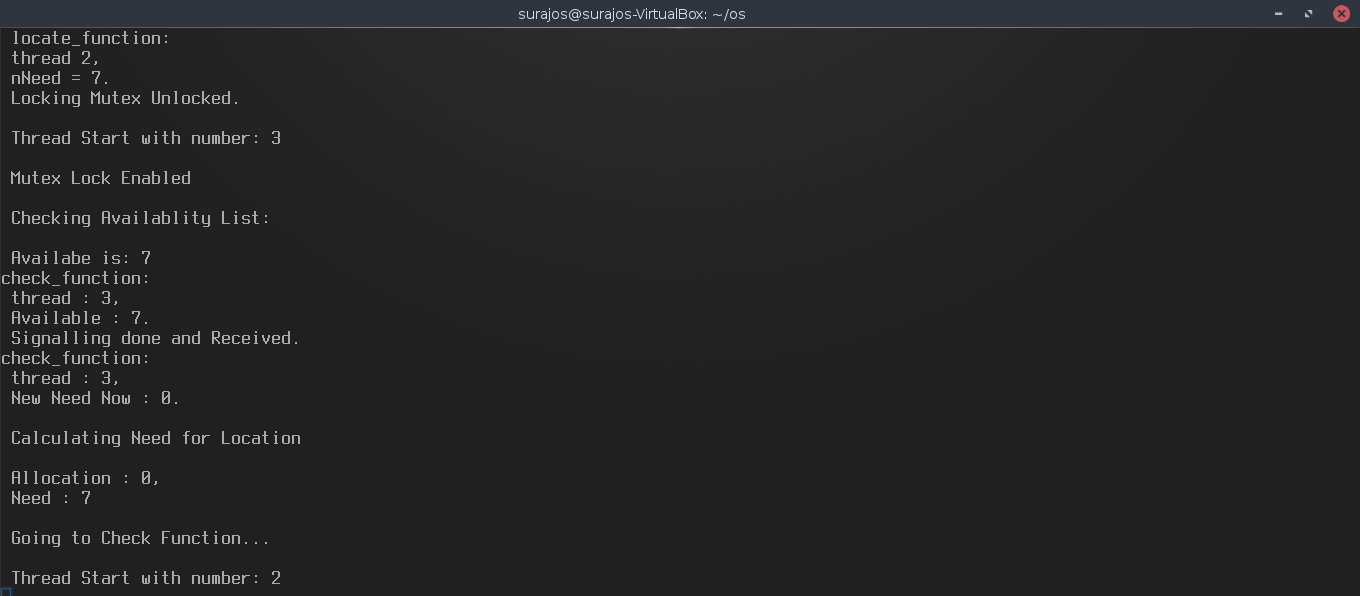
**1.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Process | Allocation   |  |  |  | | --- | --- | --- | | A | B | C | | Max\_Resource   |  |  |  | | --- | --- | --- | | A | B | C | | Need   |  |  |  | | --- | --- | --- | | A | B | C | | Avail   |  |  |  | | --- | --- | --- | | A | B | C | |
| P0 | 0 1 0 | 7 5 3 | 7 4 3 | 3 3 2 |
| P1 | 2 0 0 | 3 2 2 | 1 2 2 |  |
| P2 | 3 0 2 | 9 0 2 | 6 0 0 |  |

 8 3 4 is the final Allocation data of the Allocation Table

**2.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Process | Allocation   |  |  |  | | --- | --- | --- | | A | B | C | | Max\_Resource   |  |  |  | | --- | --- | --- | | A | B | C | | Need   |  |  |  | | --- | --- | --- | | A | B | C | | Avail   |  |  |  | | --- | --- | --- | | A | B | C | |
| P0 | 0 1 0 | 7 5 3 | 7 4 3 | 3 3 2 |
| P1 | 2 0 1 | 6 2 2 | 4 2 1 |  |
| P2 | 3 0 2 | 9 0 2 | 6 0 0 |  |

 Request Cant be granted . since the request is leading to dead lock state.

**8. Have you made minimum 5 revisions of solution on GitHub?**

Solution : Yes, 5 revisions have been made.

**GitHub link** **:** https://github.com/surajkumardas/Bankers-s-Algorithm