



LOVELY
PROFESSIONAL
UNIVERSITY

School of computer Science

Progress Report of Simulation based assignment on

[Multithreads program that implements Banker's Algorithm](#)

Submitted to: - Lovely Professional University

Faculty name: - Cherry Khosla

Submitted by: -

Name: - Md Amjad Ansari

Reg No: - 12114768

Roll No: - RK21SBB42

Section: - K21SB

Group: - 2

GitHub account: - mrAmjad123

GitHub URL: -

<https://github.com/mrAmjad123/Simulation Based Assignment CSE316 OS>

Chapter	Content
1.	INTRODUCTION
2.	Problem Definition
3.	Methodology adopted to solve the problem
4.	Algorithm/Steps
5.	Implementation
6.	Results
7.	Conclusion

1. INTRODUCTION: - The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

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. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total

money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it. In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

2. Problem Definition: - The Banker's algorithm is a resource allocation and deadlock avoidance algorithm that is used in operating systems. The main objective of the Banker's algorithm is to determine whether a particular request for resources by a process can be granted safely or not. The algorithm is based on the assumption that a system has a finite number of resources that can be allocated to multiple processes.

The implementation of the Banker's algorithm requires the ability to manage and manipulate matrices, as well as the ability to track the allocation and release of resources. The main challenge is to ensure that resources are allocated in a way that avoids deadlocks and ensures the safety of the system.

3. Methodology adopted to solve the problem: - The Banker's algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems to determine if the allocation of requested resources to a process can be granted safely or not. The algorithm is based on the assumption that a system has a finite number of resources that can be allocated to multiple processes.

- Read in the current state of the system, including the number of processes, the number of resource types, and

the current allocation and maximum resource requirements for each process.

- Calculate the available resources by subtracting the current resource allocation from the total resources.
- Create two arrays: work and finish. The work array is initialized to the available resources, and the finish array is initialized to false for all processes.
- Iterate through the processes and mark any process that can be completed with the current available resources by comparing the remaining needed resources for that process with the work array.
- Repeat step 4 and 5 until all processes are marked as finished or there are no more processes that can be completed.
- Repeat step 4 and 5 until all processes are marked as finished or there are no more processes that can be completed.
- If all processes are marked as finished, then the system is in a safe state, and the requested resources can be allocated. If not, then the system is in an unsafe state, and the requested resources cannot be allocated without the risk of causing a deadlock.

It's important to note that the Banker's algorithm can only prevent deadlocks if the resource allocation is managed properly. The algorithm assumes that the maximum resource requirements for each process are known in advance, and the system can allocate resources in a way that prevents a process from acquiring resources that are already allocated to another process. Additionally, the algorithm assumes that all processes will release their allocated resources once they are finished using them.

4. ALGORITHM: -Banker's algorithm consists of two main algorithm

- Safety algorithm.
- Resource request algorithm

1. Safety algorithm: - The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length 'm' and 'n' respectively.

Initialize: Work = Available

Finish[i] = false; for i=1, 2, 3, 4....n

2) Find an i such that both a)

Finish[i] = false

b) Need_i ≤ Work

if no such i exists goto step (4) 3)

Work = Work + Allocation[i]

Finish[i] = true goto

step (2)

4) if Finish [i] = true for all i then the system is in a safe state

2. Resource request algorithm: - Let Request_i be the request array for process P_i. Request_i [j] = k means

process P_i wants k instances of resource type R_j . When a request for resources is made by process P_i , the following actions are taken:

1) If $Request_i \leq Need_i$

Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i \leq Available$

Goto step (3); otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$Available = Available - Request_i$

$Allocation_i = Allocation_i + Request_i$

$Need_i = Need_i - Request_i$

5. Implementation: -

```
# 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]);
    }
    printf("\n");

    // maximum
    required
    resources
    for (int i=0; i < nProcesses; i++) {
        printf("\nMaximum resource required by process %d (R1 R2 ...)? ", i +
1);
        for (int j=0; j < nResources; j++)

```

```

scanf("%d", & maxRequired[i][j]);
}
printf("\n");

// calculate need matrix
need = (int ** )malloc(nProcesses * sizeof( * need));
for (int i=0; i < nProcesses; i++)
need[i] = (int * )malloc(nResources * sizeof( ** need));

for (int i=0; i < nProcesses; i++)
for (int j=0; j < nResources; j++)
need[i][j] = maxRequired[i][j] - allocated[i][j];

// get safe sequence
safeSeq = (int * )malloc(nProcesses * sizeof( * safeSeq));
for (int i=0; i < nProcesses; i++) safeSeq[i] = -1;

if (!getSafeSeq()) {
printf("\nUnsafe State! The processes leads the system to a unsafe
state.\n\n");
exit(-1);
}

printf("\n\nSafe Sequence Found : ");
for (int i=0; i < nProcesses; i++) {
printf("%-3d", safeSeq[i]+1);
}

printf("\nExecuting Processes...\n\n");
sleep(1);

// run threads
pthread_t processes[nProcesses];
pthread_attr_t attr;
pthread_attr_init( & attr);

int processNumber[nProcesses];
for (int i=0; i < nProcesses; i++) processNumber[i] = i;

for (int i=0; i < nProcesses; i++)
pthread_create( & processes[i], & attr, processCode, (void * )( &
processNumber[i]));

for (int i=0; i < nProcesses; i++)
pthread_join(processes[i], NULL);

printf("\nAll Processes Finished\n");

// free resources
free(resources);
for (int i=0; i < nProcesses; i++) {
free(allocated[i]);
free(maxRequired[i]);
free(need[i]);
}
free(allocated);
free(maxRequired);
free(need);
free(safeSeq);
}

```



```

bool
getSafeSeq()
{
    // get
    safe
    sequence
    int
    tempRes[nResources];
    for (int i=0; i < nResources; i++)
        tempRes[i] = resources[i];

    bool
    finished[nProcesses];
    for (int i=0; i < nProcesses; i++)
        finished[i] = false;
    int
    nfinished = 0;
    while (nfinished < nProcesses) {
        bool safe = false;

        for (int i=0; i < nProcesses; i++) {
            if (!finished[i]) {
                bool possible = true;

                for (int j=0; j < nResources; j++)
                    if (need[i][j] > tempRes[j]) {
                        possible = false;
                        break;
                    }

                if (possible) {
                    for (int j=0; j < nResources; j++)
                        tempRes[j] += allocated[i][j];
                    safeSeq[nfinished] = i;
                    finished[i] = true;
                    ++nfinished;
                    safe = true;
                }
            }
        }

        if (!safe) {
            for (int k=0; k < nProcesses; k++) safeSeq[k] = -1;
            return false; // no
            safe
            sequence
            found
        }
        }
    return true; // safe
    sequence
    found
}

// process
code
void * processCode(void * arg)
{
    int

```

```

p = *((int *) arg);

// lock
resources
pthread_mutex_lock( & lockResources);

// condition
check
while (p != safeSeq[nProcessRan])
    pthread_cond_wait( & condition, & lockResources);

    // process
    printf("\n--> Process %d", p + 1);
    printf("\n\tAllocated : ");
    for (int i=0; i < nResources; i++)
        printf("%3d", allocated[p][i]);

    printf("\n\tNeeded : ");
    for (int i=0; i < nResources; i++)
        printf("%3d", need[p][i]);

    printf("\n\tAvailable : ");
    for (int i=0; i < nResources; i++)
        printf("%3d", resources[i]);

    printf("\n");
    sleep(1);

    printf("\tResource Allocated!");
    printf("\n");
    sleep(1);
    printf("\tProcess Code Running...");
    printf("\n");
    sleep(rand() % 3 + 2); // process
    code
    printf("\tProcess Code Completed...");
    printf("\n");
    sleep(1);
    printf("\tProcess Releasing Resource...");
    printf("\n");
    sleep(1);
    printf("\tResource Released!");

    for (int i=0; i < nResources; i++)
        resources[i] += allocated[p][i];

    printf("\n\tNow Available : ");
    for (int i=0; i < nResources; i++)
        printf("%3d", resources[i]);
    printf("\n\n");

    sleep(1);

    // condition
    broadcast
    nProcessRan + +;
    pthread_cond_broadcast( & condition);
    pthread_mutex_unlock( & lockResources);
    pthread_exit(NULL);

```

```
}
```

6. Result: - The result of the banker's algorithm is to determine whether a requested resource allocation can be granted to a process without causing a deadlock.

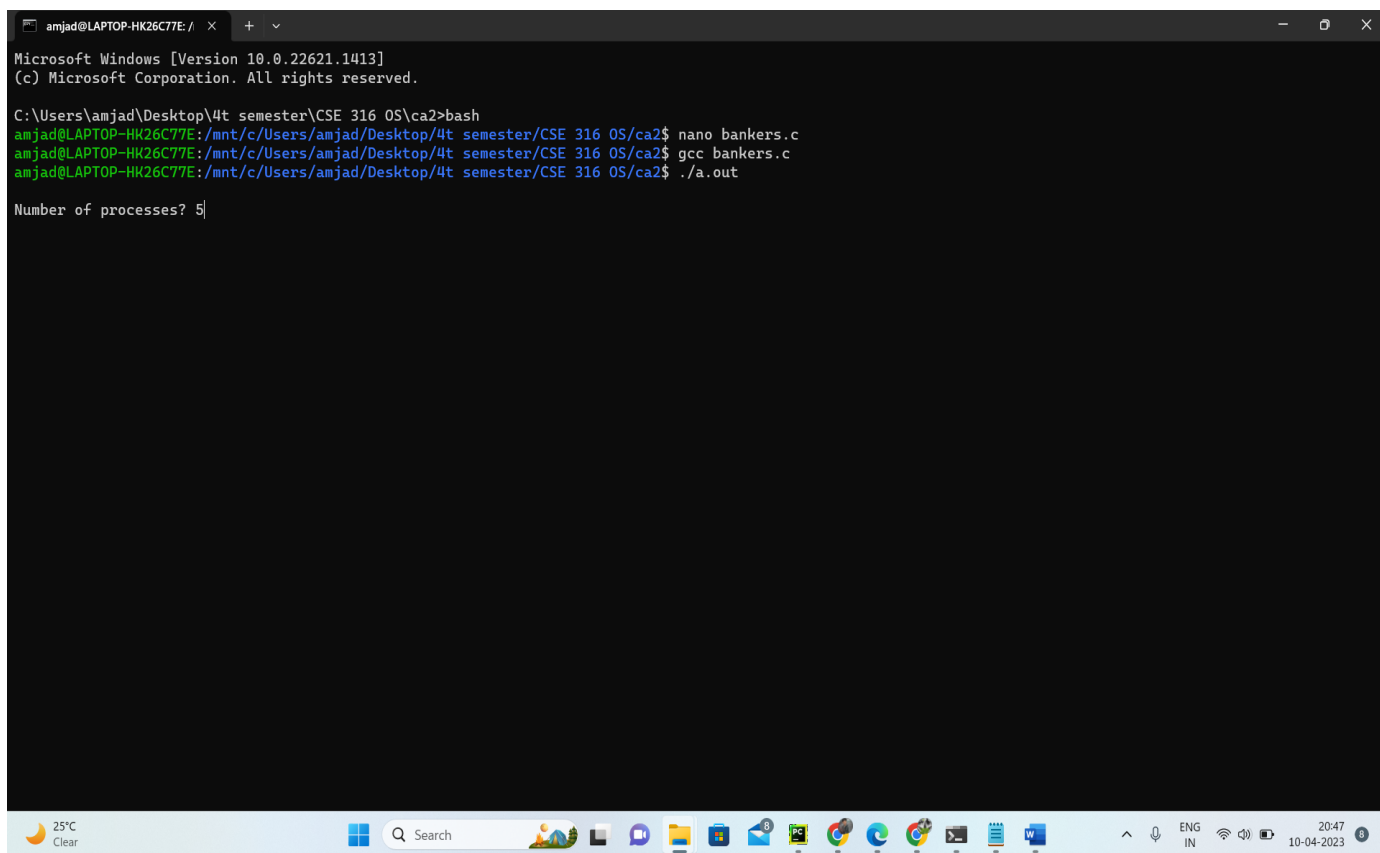
In the provided code, the banker's algorithm is used to determine if a safe sequence of process execution can be found given the current state of the system (i.e., available resources, allocated resources, and maximum required resources). If a safe sequence is found, the processes are executed in that order, otherwise, the system is deemed to be in an unsafe state and the program terminates.

If a safe sequence is found and the processes are executed, the program output will show the order in which the processes are executed. If a safe sequence is not found, the program will output a message indicating that the system is in an unsafe state.

In general, the result of the banker's algorithm is to ensure that resources are allocated in a way that does not result in a deadlock or a state where no further progress can be made.

Snapshot of the of the following program is: -

n



The screenshot shows a Windows terminal window with the following content:

```
amjad@LAPTOP-HK26C77E: / X + v
Microsoft Windows [Version 10.0.22621.1413]
(c) Microsoft Corporation. All rights reserved.

C:\Users\amjad\Desktop\4t semester\CSE 316 OS\ca2>bash
amjad@LAPTOP-HK26C77E: /mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ nano bankers.c
amjad@LAPTOP-HK26C77E: /mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ gcc bankers.c
amjad@LAPTOP-HK26C77E: /mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ ./a.out

Number of processes? 5
```

The terminal window has a title bar with the text "amjad@LAPTOP-HK26C77E: / X + v". The Windows taskbar at the bottom shows the date and time as "10-04-2023 20:47" and the language as "ENG IN".

In above picture we have to enter the number of process.

```
amjad@LAPTOP-HK26C77E: / X + v
Microsoft Windows [Version 10.0.22621.1413]
(c) Microsoft Corporation. All rights reserved.

C:\Users\amjad\Desktop\4t semester\CSE 316 OS\ca2>bash
amjad@LAPTOP-HK26C77E: /mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ nano bankers.c
amjad@LAPTOP-HK26C77E: /mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ gcc bankers.c
amjad@LAPTOP-HK26C77E: /mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ ./a.out

Number of processes? 5
Number of resources? 3|
```

Now we have to enter the total number of resources that are available in our system.

```
amjad@LAPTOP-HK26C77E: / X + v
Microsoft Windows [Version 10.0.22621.1413]
(c) Microsoft Corporation. All rights reserved.

C:\Users\amjad\Desktop\4t semester\CSE 316 OS\ca2>bash
amjad@LAPTOP-HK26C77E: /mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ nano bankers.c
amjad@LAPTOP-HK26C77E: /mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ gcc bankers.c
amjad@LAPTOP-HK26C77E: /mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ ./a.out

Number of processes? 5
Number of resources? 3
Currently Available resources (R1 R2 ...)? 3 3 2

Resource allocated to process 1 (R1 R2 ...)? 0 1 0
Resource allocated to process 2 (R1 R2 ...)? 2 0 0
Resource allocated to process 3 (R1 R2 ...)? 0 0 2
Resource allocated to process 4 (R1 R2 ...)? 1 1 2
Resource allocated to process 5 (R1 R2 ...)? 3 0 2

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

Now we have to enter how many instances of each resource has been allocated to each process.

```
amjad@LAPTOP-HK26C77E: / x + v
Microsoft Windows [Version 10.0.22621.1413]
(c) Microsoft Corporation. All rights reserved.

C:\Users\amjad\Desktop\4t semester\CSE 316 OS\ca2>bash
amjad@LAPTOP-HK26C77E:/mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ nano bankers.c
amjad@LAPTOP-HK26C77E:/mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ gcc bankers.c
amjad@LAPTOP-HK26C77E:/mnt/c/Users/amjad/Desktop/4t semester/CSE 316 OS/ca2$ ./a.out

Number of processes? 5
Number of resources? 3
Currently Available resources (R1 R2 ...)? 3 3 2

Resource allocated to process 1 (R1 R2 ...)? 0 1 0
Resource allocated to process 2 (R1 R2 ...)? 2 0 0
Resource allocated to process 3 (R1 R2 ...)? 0 0 2
Resource allocated to process 4 (R1 R2 ...)? 1 1 2
Resource allocated to process 5 (R1 R2 ...)? 3 0 2

Maximum resource required by process 1 (R1 R2 ...)? 7 5 3
Maximum resource required by process 2 (R1 R2 ...)? 4 3 3
Maximum resource required by process 3 (R1 R2 ...)? 2 2 2
Maximum resource required by process 4 (R1 R2 ...)? 6 0 2
Maximum resource required by process 5 (R1 R2 ...)? 3 2 2
```

Now we have to enter the maximum resource are required by each process.

```
amjad@LAPTOP-HK26C77E: / x + v

Safe Sequence Found : 3 5 2 4 1
Executing Processes...

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

--> Process 5
    Allocated : 3 0 2
    Needed    : 0 2 0
    Available  : 3 3 4
    Resource Allocated!
    Process Code Running...
    Process Code Completed...
    Process Releasing Resource...
    Resource Released!
    Now Available : 6 3 6

--> Process 2
    Allocated : 2 0 0
    Needed    : 2 3 3
    Available  : 6 3 6
    Resource Allocated!
    Process Code Running...
    Process Code Completed...
    Process Releasing Resource...
    Resource Released!
```

We have found the safe sequence of the process to be executed so that there will not be occur deadlock in the system.

```
amjad@LAPTOP-HK26C77E: ~$
--> Process 2
Allocated : 2 0 0
Needed : 2 3 3
Available : 6 3 6
Resource Allocated!
Process Code Running...
Process Code Completed...
Process Releasing Resource...
Resource Released!
Now Available : 8 3 6

--> Process 4
Allocated : 1 1 2
Needed : 5 -1 0
Available : 8 3 6
Resource Allocated!
Process Code Running...
Process Code Completed...
Process Releasing Resource...
Resource Released!
Now Available : 9 4 8

--> Process 1
Allocated : 0 1 0
Needed : 7 4 3
Available : 9 4 8
Resource Allocated!
Process Code Running...
Process Code Completed...
Process Releasing Resource...
Resource Released!
Now Available : 9 5 8

All Processes Finished
amjad@LAPTOP-HK26C77E: /mnt/c/Users/amjad/Desktop/4t_semester/CSE_316_OS/ca2$
```

All the process has been executed successfully and safely.

Conclusion: - Now the Algorithm has been successfully executed, it means that the system has gone through a process of resource allocation in a safe and efficient manner. The algorithm is designed to prevent deadlock, which is a situation where multiple processes are waiting for resources that are being held by other processes, resulting in a standstill where none of the processes can proceed.

By successfully executing the Banker's Algorithm, it can be concluded that the system has allocated resources in a way that guarantees that all processes will eventually be able to

complete their execution without getting stuck in a deadlock. This ensures that the system is running smoothly and that all processes can proceed as intended, without any interruptions or delays caused by resource allocation issues.

In summary, the successful execution of the Banker's Algorithm is an important milestone in ensuring the efficient and safe operation of a computer system, and it is a sign that the system is capable of handling resource allocation in a way that avoids potential problems such as deadlock.

Thank You