JEEMECS (Journal of Electrical Engineering, Mechatronic and Computer Science ISSN 2614-4859
Vol. 6, No. 1, February 2023, pp. xx-xx

Banker's Algorithm Optimization to Dynamically Avoid Deadlock in Operating System

Herlambang Rafli Wicaksono [a,1,*], Herisa Pratama Nur Baeti [a,2], Yasmin Putri Salma [a,3], Aqwam Rosadi Kardian [b,4]

[a] Politeknik Siber dan Sandi Negara, Jl. H. Usa, Ciseeng, Kabupaten Bogor, West Java 16120, Indonesia
[b] Sekolah Tinggi Manajemen Informatika dan Komputer Jakarta STI&K, Jl. Bri Radio Dalam No.17, Kebayoran Baru, South Jakarta, Jakarta 12140, Indonesia
[1] raflihw1@gmail.com [*]; [2] herisa.651@gmail.com; [3] yasminsalma2601@gmail.com; [4] aqwam@staff.jak-stik.ac.id
[*] corresponding author

ABSTRACT

Keywords
Operating System
Deadlock
Banker's Algorithm

Operating systems hold the responsibility to ensure a computer works as it was intended, including that every resource in a computer is managed and used by any process needing them. Bad resource allocation can lead to a condition where the resources cannot be accessed because they are still used by a frozen process, known as deadlock. Deadlock avoidance in the operating system is usually done using a banker's algorithm. The current algorithm still possesses limitations, including the need to determine the number of processes before starting the calculation. This paper proposes an optimized banker's algorithm that can be intercepted in the middle of execution in case of a new process requests some resources so deadlock can be avoided dynamically.

1. Introduction

Computers are widely used in every aspect of modern life since they made every task significantly easier and faster. Computers are made in various architectures and designs, which are unique and proprietary to each manufacturer. This differently structured hardware needs to be able to run the standard and most used function, leading to the need for operating systems. Operating systems provide the ability for each different architecture to run the standard need notably to manage resources in a computer, including memory management, process management, input-output management, secondary-storage management, and file management.

In some environment, a finite number of resources may be requested by several processes. This is a critical issue because the resources is limited and might not be shareable. Usually when a process requests for unavailable resources; it enters a waiting state. In some cases, the waiting state can last forever, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. The Kansas legislature draws a deadlock illustration early in the 20th century which said, in part: "When two trains approach each other at a crossing, both shall come to a full stop, and neither shall start up again until the other has gone." Deadlock will happen if it met conditions as follow: (1) a resource that can only be used by one process at a time (mutual exclusion), (2) request for new resources held by other processes can be done by processes already holding resources (hold and wait/wait-for), (3) resource a process cannot be forced to release resources it held (no preemption), and (4) two or more processes waits for a resource that another process hold forming a circular chain (circular wait). A method to avoid deadlock is called deadlock avoidance [1][2][3][4].

The banker's algorithm, an algorithm proposed by Edsger W. Dijkstra, is an algorithm inspired by bank modeling related to customer credit applications. This algorithm is used to avoid deadlocks and allocate resources safely to every process in a computer system [5]. When a new process is created in a computer system the process must provide the operating system with all kinds of information such as future access requests and delays for computer resources. Based on this information the operating system decides whether to execute a sequence of processes or wait until a deadlock occurs in the

system [6]. Hence it is also known as a deadlock detection algorithm or deadlock detection in the operating system. Banker's algorithm still has some limitations, such as it needs to know how many resources each process can request. Most systems lack information about resources and processes that make Banker's algorithm impossible [7]. Most systems also have a dynamically changing number of processes, so it is not practical to assume a constant number of processes.

2. The Proposed Algorithm

2.1. Algorithm Description

The algorithm studied in this paper is an optimized algorithm for the existing banker's algorithm. When a new process request resource, the banker's algorithm will decide whether the request is granted or not based on the current usage of resources by other running processes. It will also generate a safe sequence that will provide an order of process execution without deadlock [8][9]. If there is another request from another process during the execution of this algorithm, the request must wait until the algorithm returns its output and be executed again. The proposed algorithm is focused to manage this problem by making the algorithm dynamic, where the algorithm can be intercepted in case there is another resource request in the middle of its execution.

2.2. Data Structures

Initially, there will be M running processes and N resource types. The main matrices in the banker's algorithm are the maximum demand matrix Max[M,N], the assigned matrix Allocation[M,N], and the demand matrix Need[M,N]. Their relationship is the demand matrix Need[M,N]= Max[M,N]-Allocation[M,N] [10]. The complete data structure of the optimized banker's algorithm is described as follows:

- Maximum demand matrix Max[M,N], Max [i][j] in the operating system environment is expressed as the maximum demand quantity of Rj resources for process Pi, where the largest demand matrix Max[M,N] is an M a matrix of rows N;
- Allocation matrix Allocation[M,N], Allocation [i][j] in the operating system environment indicates that the number of Rj resources has been obtained for the process Pi, where the allocation matrix Allocation[M,N] is an M row N a matrix of columns;
- Demand matrix Need[M,N], Need[i][j] in the operating system environment indicates the number of R j resources required for the process Pi, where the demand matrix Need[M,N] is an M row N a matrix of columns;
- The available resource vector Available, one containing M elements Array vector, the value of the array vector represents the number of resources available for a class; and
- Finish[i] indicates whether the resource has an identification vector, and the Linux operating system environment indicates whether the system has sufficient resources to allocate to the process Pi.

2.3. Algorithm Flowchart and Pseudocode

The proposed version of the banker's algorithm is described as follows:

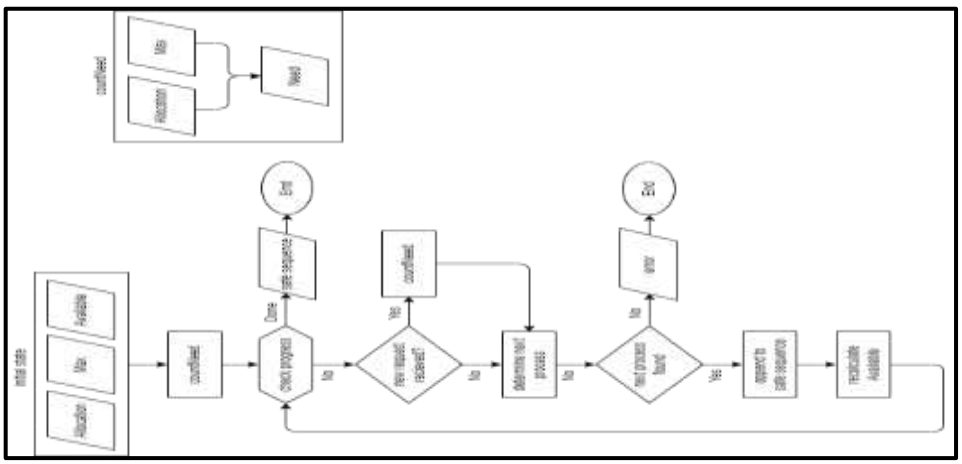

Fig 1. Algorithm flowchart

The proposed algorithm will receive the initial state of Allocation[M,N], Max[M,N], and Available[N] as the input. It will calculate the initial Need[M,N] and start the banker's original algorithm. The difference is before the execution of each loop, it will check whether a new request is received. If a new request is received, it will recalculate the Need[M,N] and continue the execution as it should. The code implementation can be divided as follows:

- Function to count the Need[M,N] from Max[M,N] and Allocation[M,N], described as follow:

```
DEFINE FUNCTION countNeed(Allocation, Max, N):

    SET Need TO [[ 0 FOR i IN range(N)] FOR i IN ENUMERATE(Allocation)]
    FOR i, x IN ENUMERATE(Allocation):
        FOR j IN range(N):
            SET Need[i][j] TO Max[i][j] - Allocation[i][j]

    RETURN Need
```

Fig 2. Pseudocode of the function to count need

This function will count the Need matrix based on Max and Allocation matrix. It iterates through and determines the Need value of each process.

- Function to run the optimized banker's algorithm described as follows:

```
DEFINE FUNCTION bankerLoop(Allocation, N, Available, Need, Max):

    SET M TO len(Allocation)
    SET Finish TO [False]*M
    SET safeSequence TO []

    SET loopWillStuck TO False
    WHILE False IN Finish and not loopWillStuck:

        IF NEW_RESOURCE_REQUEST_RECIEVED:
            Allocation.append([0]*N)
            Max.append(NEW_RESOURCE_REQUEST_MAX)
            SET Need TO countNeed(Allocation, Max, N)
            SET M TO M + 1
            Finish.append(False)

        SET loopWillStuck TO True
        FOR i IN range(M):
            IF (Finish[i] EQUALS false):
                SET flag TO 0
                FOR j IN range(N):
                    IF (Need[i][j] > Available[j]):
                        SET flag TO 1
                        break

                IF (flag EQUALS 0):
                    safeSequence.append(i)
                    FOR j IN range(N):
                        Available[j] += Allocation[i][j]

                    SET Finish[i] TO True
                    SET loopWillStuck TO False

        IF loopWillStuck:
            RETURN "error"

    RETURN safeSequence
```

Fig 3. Pseudocode of the function to execute the optimized banker's algorithm

This function will execute the optimized banker's algorithm and return a safe sequence to execute corresponding processes. It will iterate, check, and compare the work with the available resources to determine if the process can be executed. On every iteration, the function will listen to new request and recalculate the need if new request is received. Thus, it can be instantly determined whether the request is granted or not.

The functions can be executed when a new resource request received as follow:

```
SET N
SET Allocation
SET M TO len(Allocation)
SET Max
SET Available

SET Need TO countNeed(Allocation, Max, N)
SET safeSequence TO bankerLoop(Allocation, N, Available, Need, Max)

IF safeSequence EQUALS 'error':
    OUTPUT('Impossible to create safe sequence')
    exit()

OUTPUT(safeSequence)
```

Fig 4. Pseudocode of the main code

The main code will set the initial state and count the initial resource need. The result will be passed to the optimized banker's algorithm function. If the function returns a safe sequence, then the processes can be executed safely without a deadlock.

3. Method

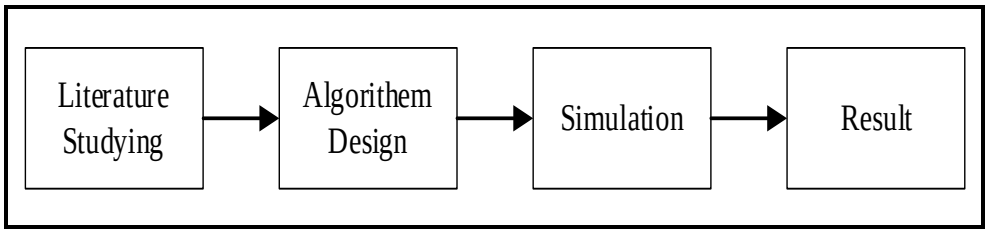

Fig 5. Research framework

This paper will propose an optimized version of banker's algorithm based on the limitation it currently had. The proposed algorithm design is described in Fig. 1. The proposed algorithm is simulated using python programming language on Windows 11 operating system. The simulation will use tables to represent the calculating process of the Need of each process and the state of the optimized banker's loop algorithm at certain time.

4. Results and Discussion

The proposed algorithm will be executed as follow: Assume that there are three (N) resources of A, B, and C in the operating system. The number of Class A resources is 10, the number of Class B resources is five, and the number of Class C resources is five. Initially, four (M) processes named p0, p1, p2, and p3 are running. The state at T0 is shown in Table 3,

Table 1. Resource Need at T0

Process	System Resource			
	Max	***Allocation***	***Need***	***Available***
	A B C	***A B C***	***A B C***	***A B C***
p0	7 5 3	0 1 0	7 4 3	3 3 2
p1	3 2 2	2 0 0	1 2 2	
p2	9 0 2	3 0 2	6 0 0	
p3	2 2 2	2 1 1	0 1 1	

In general, operating systems will execute the standard banker's algorithm to determine if a safe sequence can be generated. The algorithm will loop M times and generate a safe sequence as follow:

Table 2. System Resource State at T4 using Standard Banker's Algorithm

Process	System Resource				Finish
	Work *A B C*	*Allocation* *A B C*	*Need* *A B C*	*Available* *A B C*	
p1	3 3 2	2 0 0	1 2 2	5 3 2	T
p3	5 3 2	2 1 1	0 1 1	7 4 3	T
p0	7 4 3	0 1 0	7 4 3	7 5 3	T
p2	7 5 3	3 0 2	6 0 0	10 5 5	T

The generated safe sequence will be p1 → p3 → p0 → p2 and all processes can be executed without a deadlock. If a new resource request is accepted in the middle of execution namely at T2, the request must wait until T4 and start new execution with new data. Using the proposed algorithm, the execution can be intercepted and injected by the new request as follow:

Table 3. System Resource State at T2

Process	System Resource				Finish
	Work *A B C*	*Allocation* *A B C*	*Need* *A B C*	*Available* *A B C*	
p1	3 3 2	2 0 0	1 2 2	5 3 2	T
p3	5 3 2	2 1 1	0 1 1	7 4 3	T

Assume a new process p4 request resource (4, 3, 3) at this state, the Need matrix will be recalculated as follow:

Table 4. Resource Need at T2

Process	System Resource			
	Max *A B C*	*Allocation* *A B C*	*Need* *A B C*	*Available* *A B C*
p0	7 5 3	0 1 0	7 4 3	
p2	9 0 2	3 0 2	6 0 0	7 4 3
p4	4 3 3	0 0 0	4 3 3	

Then the execution can be continued as follow:

Table 5. System Resource State at T5 using Optimized Banker's Algorithm

Process	*Work* *A B C*	System Resource			Finish
		Allocation *A B C*	*Need* *A B C*	*Available* *A B C*	
p1	3 3 2	2 0 0	1 2 2	5 3 2	T
p3	5 3 2	2 1 1	0 1 1	7 4 3	T
p0	7 4 3	0 1 0	7 4 3	7 5 3	T
p2	7 5 3	3 0 2	6 0 0	10 5 5	T
p4	10 5 5	0 0 0	4 3 3	10 5 5	T

The generated safe sequence will be p1 → p3 → p0 → p2 → p4 and all processes can be executed without a deadlock.

5. Conclusion

This paper proposes an optimized version of the existing banker's algorithm to dynamically avoid deadlock in the operating system. When a new process makes a new resource request in the middle of the banker's algorithm execution, it must wait until the banker's algorithm returns a value and start another one with new data. The banker's algorithm is improved for this problem. In the middle of the banker's algorithm execution, a new process can intercept the execution and inject a new resource

request, then the execution will continue to check and generate a safe sequence. After obtaining the security sequence, deadlock avoidance in the operating system is guaranteed. System simulation and experimental verification show that this algorithm ensures that the system will not deadlock and improve the reliability of the system.

References

[1] J. Ezpeleta, F. Tricas, F. García-Vallés, and J. M. Colom, " A Banker's Solution for Deadlock Avoidance in FMS With Flexible Routing and Multiresource States," Ieee Transactions on Robotics And Automation, vol 18, August 2002.

[2] Ms. Kshipra Dixit and Dr. Ajay Khuteta, "A Dynamic and Improved Implementation of Banker's Algorithm," International Journal on Recent and Innovation Trends in Computing and Communication, vol.5, pp. 45-49, Agustus 2017.

[3] Yang Wang and Paul Lu, "Maximizing Active Storage Resources with Deadlock Avoidance in Workflow-Based Computations," IEEE Transactions on Computers, Vol. 62 No. 11, November 2013.

[4] M. Begum, O. Faruque, M. W. Rahman Miah and B. Chandra Das, "An Improved Safety Detection Algorithm Towards Deadlock Avoidance," 2020 IEEE 10th Symposium on Computer Applications & Industrial Electronics (ISCAIE), , pp. 73-78, 2020, doi: 10.1109/ISCAIE47305.2020.9108818.

[5] Markus Bangun, "Simulasi Algoritma Banker pada Sistem Antrian," Jurnal Mahajana Informasi, Vol.2 No 2, 2017.

[6] V. Bobanac and S. Bogdan, "Routing and scheduling in Multi-AGV systems based on dynamic banker algorithm," 2008 16th Mediterranean Conference on Control and Automation, pp. 1168-1173, 2008, doi: 10.1109/MED.2008.4602057.

[7] M. Žarnay and F. T. García, "Enhancing banker's algorithm for avoiding deadlocks in systems with non-sequential processes," Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), pp. 1-8, 2014, doi: 10.1109/ETFA.2014.7005149.

[8] Li Jiang, "Process Security Sequence Improvement Algorithm Based on Banker Algorithm," IOP Conf. Series: Journal of Physics: Conf. Series 1237, 2019.

[9] D. Song, Y. Li and T. Song, "Modified Banker's algorithm with dynamically release resources," 2021 International Conference on Communications, Information System and Computer Engineering (CISCE), pp. 566-569, 2021, doi: 10.1109/CISCE52179.2021.9445935.

[10] K. T. S. Kasthuriarachchi and U. U. S. K. Rajapaksha, "Design of auxiliary simulator for analysing the deadlock occurrence using Banker's algorithm," 2015 Fifteenth International Conference on Advances in ICT for Emerging Regions (ICTer), pp. 265-265, 2015, doi: 10.1109/ICTER.2015.7377698.