GROUP - 34

1741016 Kausha Vora

1741048 Nimil Shah

BANKER'S ALGORITHM

TABLE OF CONTENTS:

NO.	Objectives	Page No
1	Introduction	2
2	Deadlock Avoidance	2
3	Bankers algorithm	3
4	Implementation Of Banker's Algorithm in our code	5
5	Banker's Algorithm Example	6
6	Conclusion	10

1. INTRODUCTION

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

Most current operating systems do not provide deadlock-prevention facilities, but such features will probably be added soon. Deadlock problems can only become more common, given current trends, including larger numbers of processes, multithreaded programs, many more resources within a system, and the emphasis on long-lived file and database servers rather than batch systems.

2. DEADLOCK AVOIDANCE

Deadlock avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes. The following are the methods of deadlock avoidance:

1. Safe state
2. Resource Allocation Graph Algorithm
3. Banker's Algorithm

3. BANKER'S ALGORITHM

3.1. Introduction

Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes a "safe state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue. In an operating system, deadlock is a state in which two or more processes are "stuck" in a circular wait state. All deadlocked processes are waiting for resources held by other processes. Because most systems are non-preemptive (that is, will not take resources held by a process away from it), and employ a hold and wait method for dealing with system resources (that is, once a process gets a certain resource it will not give it up voluntarily), deadlock is a dangerous state that can cause poor system performance.

3.2. Goal

The goal of this algorithm is to handle all the requests without entering into the unsafe state.

3.3. Working

A state is considered safe if it is possible for all processes to finish executing (terminate). Since the system cannot know when a process will terminate, or how many resources it will have requested by then, the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward. Also, if a process terminates without

acquiring its maximum resource it only makes it easier on the system.

A safe state is considered to be the decision maker if it is going to process ready queue. Safe State ensures the Security. Given that assumption, the algorithm determines if a state is safe by trying to find a hypothetical set of requests by the processes that would allow each to acquire its maximum resources and then terminate (returning its resources to the system). Any state where no such set exists is an unsafe state

The Banker's algorithm is run by the operating system whenever a process requests resources. The algorithm prevents deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock would occur).

When a new process enters a system, it must declare the maximum number of instances of each resource type that may not exceed the total number of resources in the system.

4. IMPLEMENTATION OF BANKER'S ALGORITHM IN OUR CODE

4.1. Variables Used

1. Total Number Of Processes: numberof_processes
2. Total Number Of Resource Types: numberof_resources
3. Instances Of Every Resource Type: totalinstances [ ]
4. Resources Of Each Type Allocated To Every Process: allocated_resources [ ] [ ]
5. Available Instances Of Every Resource: available_resources [ ]
6. Maximum Required Resource Of Each Type For Every Process After First Allocation: maxRequired_resources [ ] [ ]
7. Resources Needed Of Each Type For Every Process: needof_resources [ ] [ ]

4.2. Conditions To Be Satisfied

Resources may be allocated to a process only if it satisfies the following conditions:

1. Request < = max, else set error condition as process has crossed maximum claim made by it.
2. Request < = available, else process waits till resources are available. Some of the resources that are tracked in real systems are memory, semaphores and

5. BANKER'S ALGORITHM EXAMPLE

Considering a system with five processes P_0 through P_4 and three resources of type Printers, Disk Drives, Tapes being (A, B, C). Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

Process	Allocation			Max Required			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	2	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

NOW, needof_resources[i, j] = maxRequired_resources [i, j] – allocated_resources[i, j]

Therefore, the NEED matrix will be as follows:

Process	Need		
	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0

P3	0	1	1
P4	4	3	1

5.1. Iteration - 1

Examine the Need matrix and check for the row which is less than Available vector. There are two rows that are less than the Available vector, and these processes are P1 and P3. As P1 comes first in order, therefore, it gets executed first.

Need(P1) = (1, 2, 2) < (3, 3, 2) = Available

Assuming that P1 reaches successful completion, it returns the currently allocated resources to it, thereby incrementing the Available vector.

(3, 3, 2)	Current value of Available
+ (2, 0, 0)	Allocation (P1)
` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `	
(5, 3, 2)	Updated value of Available

5.2. Iteration - 2

Examine the Need matrix again, this time comparing it with Updated Available Vector. Ignore the row for P1. There are two rows, namely of P3 and P4 which are less than Updated Available Vector. As P3 is smaller in order, therefore it is executed.

Need(P3) = (0, 1, 1) < (5, 3, 2) = Available

Assuming that P3 reaches successful completion, it returns the currently allocated resources to it, thereby incrementing the Available vector.

(5, 3, 2)	Current value of Available
+ (2, 1, 1)	Allocation (P3)

` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `

(7, 4, 3)   Updated value of Available

5.3. Iteration - 3

Examine the Need matrix again, comparing it with Updated Available Vector. Ignore the row for P1 and P3. There is only one process P4 whose Need is less than Updated Available Vector.

Need(P4) = (4, 3, 1) < (7, 4, 3) = Available

Assuming that P2 reaches successful completion, it returns the currently allocated resources to it, thereby incrementing the Available vector.

(7, 4, 3)   Current value of Available
+ (0, 0, 2)   Allocation (P4)
` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `

(7, 4, 5)   Updated value of Available

5.4. Iteration - 4

Examine the Need matrix again, comparing it with Updated Available Vector. Ignore the row for P1, P3 and P4. There are two rows, namely of P0 and P2 which are less than Updated Available Vector. As P0 is smaller in order, therefore it is executed.

Need(P0) = (7, 4, 3) < (7, 5, 4) = Available

Assuming that P0 reaches successful completion, it returns the currently allocated resources to it, thereby incrementing the Available vector.

(7, 5, 4)   Current value of Available
+ (0, 1, 1)   Allocation (P0)
` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `

(7, 5, 5)   Updated value of Available

5.5. Iteration - 5

Ignore the row for P0, P1, P3, P4. Only one process, P2 is left. Compare its Need Vector with Updated Available Vector.

Need(P2) = (6, 0, 0) < (7, 5, 5) = Available

Assuming that P4 reaches successful completion, it returns the currently allocated resources to it, thereby incrementing the Available vector.

(7, 5, 5)	Current value of Available
+ (3, 0, 2)	Allocation (P2)
(10, 5, 7)	Updated value of Available

Therefore, we get all our resources back to the initial number as 10,5,7 and all the processes complete.

Safe State Sequence is : P1 P3 P4 P0 P2

6.CONCLUSION

In an operating system, deadlock is a state in which two or more processes are "stuck" in a circular wait state. All deadlocked processes are waiting for resources held by other processes. Because most systems are non-preemptive (that is, will not take resources held by a process away from it), and employ a hold and wait method for dealing with system resources (that is, once a process gets a certain resource it will not give it up voluntarily), deadlock is a dangerous state that can cause poor system performance. The goal of this algorithm is to handle all the requests without entering into the unsafe state.