İstanbul
Bilgi Üniversitesi
LAUREATE INTERNATIONAL UNIVERSITIES

CMPE 312

Operating Systems - Final Project

Banker's Algorithm

By

Raşit Deniz Cansever (112200022)

Burak Güler (11575012)

Supervised

By

Özgür Özdemir

Faculty of Engineering and Natural Sciences

Istanbul Bilgi University

May 2020

Introduction

At the present time, there are many problems which is faced by programmers in operating system. One of the most annoying problem is deadlock. In operating system, two or more processes can lock each other for various reasons. Then system would enter an infinite loop. This problem can be illustrated in the real life. Imagine that, there are 2 person which want to enter through the door. One of them is Deniz, other person's name is Burak. Deniz requires that Burak should be the one who passes first through the door. Similarly, Burak requires that Deniz should be the one who passes first through the door. Both people do not pass through the door, before one of them pass first. In this situation, they wait forever. This situation is called deadlocks. This happens frequently in the operating systems between two or more processes. These deadlock problems can be seen in the simple semaphore examples.

Banker's algorithm is an algorithm to prevent deadlocks on resources in operating system design. Banker's algorithm is used in the banking system. That's why, people have not named it anti-deadlocks algorithm but banker's algorithm. The algorithm was developed by Dijkstra. The algorithm has 3 basic states and 2 conditions.

States to know:

1. How many resources do each process need?
2. How many resources do each process currently hold?
3. How many resources are currently available?

After knowing this information above, algorithm applies the following conditions during resource allocation:

1. Refusing the job if the requested resource (need) is bigger than max (data structure of algorithm).
2. If the requested resource (need) is bigger than available (data structure of algorithm), algorithm holds the process until the instances of resources are empty.

In the algorithm detailed above, the value which is referred as the source can be anything for the operating system. (RAM, I/O devices or runtime for real systems) Banker's algorithm also needs some data structures in order to able to follow the above. For the following data structures; 'n' is the number of processes in the system and 'm' is the number of different resources.

Available:
This is an array with m elements. Each element of the array keeps how available it is from that resource type. As an example, avail[i]=5 equation means, five instances of resource types as i, are available.

Max:
This is held by a two-dimensional array and has dimensions of n x m. The amount of discrimination made from the relevant source for each transaction is marked in the series. As an example, max[2][3]=5 equation means, the second process has the right to discrimination that 5 unit right over the third source. In other words, second process can not use more than 5 units from third resource.

Allocation:
This is a two-dimensional array, and its size is kept as n x m. It shows that how much of each resource and process uses. As an example, alloc[2][3] = 4 equation means, at that moment, the second process is using 4 units from the third resource. (Holding)

Needs:
Similarly this is held as an array in n x m dimensions. This keeps how much each process needs from each source. As an example need[2][3] = 1 equation means, second process needs 1 unit from the third resource.

The output of the algorithm's work would be in safe state or unsafe state. According to this,
If processes are likely to run and end, algorithm returns in safe state and shows the result.
If processes lock each other, algorithm returns in unsafe mode.

Algorithm does the following calculations to reach to result as safe or unsafe state.

Needs = Max - Allocation
Finish[i] = True
Available = Available + Allocation

If the algorithm works with given datas according to these calculation, it gets the result in the safe state. In this case, the processes will run successfully without deadlocks. Otherwise, specific process does not work.

Methodology

Here is the methodology, pseudo-code of the Banker's Algorithm.

1) m=3; // number of the resources
n=5; // number of the processes
Initialize available[], finished[];
finished[i] = false for i = 1,2,...n

2) Get an i such that satisfies
   a) finished[i] = false
   b) need[i] <= available[]

If no such i found, jump to step 4.

3) available[] = available[] + allocation[i]
finished[i] = true
Jump to step 2 again.

4) If finished[i] = true for all i between 1 and n,
Then the algorithm worked correctly and the system is in safety.

Simulation

Since there are 5 processes and 3 resources in the example, there are 5 finished status which all are false at the beginning. The instances of the resources are 16, 17 and 15 respectively. Also, the available instances of the resources are 2, 4 and 1 respectively at that moment.

m=3, n=5	P0	P1	P2	P3	P4	R0	R1	R2
Available						2	4	1
Finished	false	false	false	false	false			

In the code, for loop in line 64 starts from P0 to check if the need of process is greater than current availability or not. If so, P0 will wait for satisfying availability and for loop will continue with the next process.

For i = 0,
need[0] = {6,3,1}, avail[] = {2,4,1}
need[0] > avail[] => P0 waits.
finished[0] = false

P1 will face with the same result as P0 did, this is because its need is also greater than current availability.

finished[1] = false

When for loop calls P2, it will not be passed because the need of that process is smaller than the availability.

For i = 2,
need[2] = {2,1,0}, avail[] = {2,4,1}
need[2] < avail[] => P2 will enter to the safe sequence.
avail[] = avail[] + alloc[2] = {2,4,1} + {2,0,6} (in line 75)
avail[] = {4,4,7}
finished[2] = true (in line 76)

P3 will wait for the needed free instances, P4 will be entered to the safe sequence after P2.
finished[3] = false
finished[4] = true
avail[] = {5,7,7}

Since there is no non-checked processes, the loop will return to the beginning and checks P0 again.

Still, the need of P0 cannot be afforded by the resources, so P0 is going to wait again.

finished[0] = false

For now, the need of P1 can be satisfied by the resources. This is because there are more available instances at this moment.

For i = 1,
need[1] = {3,1,1}, avail[] = {5,7,7}
need[1] < avail[] => P1 will enter to the safe sequence.
avail[] = avail[] + alloc[1] = {5,7,7} + {5,4,5}
avail[] = {10,11,12}
finished[1] = true

P3 also becomes suitable for the sequence since its need is smaller than the current availability.

For i = 3,
need[3] = {1,5,6}, avail[] = {10,11,12}
need[3] < avail[] => P3 is ready to be executed.
avail[] = avail[] + alloc[3] = {10,11,12} + {5,4,2}
avail[] = {15,15,14}
finished[3] = true

Finally, P0 acquires the demanded numbers of instances from the resources.

For i = 0,
need[0] = {6,3,1}, avail[] = {15,15,14}
need[0] < avail[] => P0 will enter to the safe sequence.
finished[0] = true
avail[] = avail[] + alloc[0] = {15,15,14} + {1,2,1}
avail[] = {16,17,15}. Since there is no more process that requests to be executed, the elements of the avail[] array is equal to the instances of the resources.

m=3, n=5	P0	P1	P2	P3	P4	R0	R1	R2
Available						16	17	15
Finished	true	true	true	true	true			

The program is terminated.

Conclusion

As a conclusion, at the present time, there are so much deadlocks problems when programmers face on project with more than one processes which are used in the operating system. The Banker's Algorithm is the key of these problems. This looks sharp and solid algorithm.

As it can be seen in the simulation for Banker's Algorithm above, five processes have managed to enter to the safe sequence and be executed successfully. However, the initial process was not be able to enter it, the algorithm used loops and conditions in order to sort the processes with correct order.

As a result, Banker's algorithm is very beneficial for the safety on the operating systems.