Operating Systems Z02-54c

Banker's algorithm in operating system

Dekker's algorithm for the Critical Section Problem

Mustafa Tayyip BAYRAM 257639

Banker's Algorithm

Banker's Algorithm is a resource allocation and deadlock avoidance algorithm. This algorithm test for safety 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.

In simple terms, it checks if allocation of any resource will lead to deadlock or not, OR is it safe to allocate a resource to a process and if not then resource is not allocated to that process. Determining a safe sequence(even if there is only 1) will assure that system will not go into deadlock.

Data Structures used to implement Banker's Algorithm

Available: It is a 1-D array that tells the number of each resource type (instance of resource type) currently available. Example: Available[R1]= A, means that there are A instances of R1 resources are currently available.

Max: It is a 2-D array that tells the maximum number of each resource type required by a process for successful execution. Example: Max[P1][R1] = A, specifies that the process P1 needs a maximum of A instances of resource R1 for complete execution.

Allocation: It is a 2-D array that tells the number of types of each resource type that has been allocated to the process. Example: Allocation[P1][R1] = A, means that A instances of resource type R1 have been allocated to the process P1.

Need: It is a 2-D array that tells the number of remaining instances of each resource type required for execution. Example: Need[P1][R1]= A tells that A instances of R1 resource type are required for the execution of process P1.

The Bankers Algorithm consists of the following two algorithms:

1. Request-Resource Algorithm
2. Safety Algorithm

1. Resource- Request Algorithm

Whenever a process makes a request of the resources then this algorithm checks that if the resource can be allocated or not.

It includes three steps:

1. The algorithm checks that if the request made is valid or not. A request is valid if the number of requested resources of each resource type is less than the Need(which was declared previously by the process). If it is a valid request then step 2 is executed else aborted.
2. Here, the algorithm checks that if the number of requested instances of each resource is less than the available resources. If not then the process has to wait until sufficient resources are available else go to step 3.
3. Now, the algorithm assumes that the resources have been allocated and modifies the data structure accordingly.

```
Available = Available - Request(i)
Allocation(i) = Allocation(i) + Request(i)
Need(i) = Need(i) - Request(i)
```

After the allocation of resources, the new state formed may or may not be a safe state. So, the safety algorithm is applied to check whether the resulting state is a safe state or not.

Safe state: A safe state is a state in which all the processes can be executed in some arbitrary order with the available resources such that no deadlock occurs.

a. If it is a safe state, then the requested resources are allocated to the process in actual.
b. If the resulting state is an unsafe state then it rollbacks to the previous state and the process is asked to wait longer.

2. Safety Algorithm

The safety algorithm is applied to check whether a state is in a safe state or not.

This algorithm involves the following four steps:

1. Suppose currently all the processes are to be executed. Define two data structures as work and finish as vectors of length m(where m is the length of Available vector)and n(is the number of processes to be executed).

   ```
   Work = Available
   Finish[i] =false for i = 0, 1, … , n−1.
   ```
2. This algorithm will look for a process that has Need value less than or equal to the Work. So, in this step, we will find an index i such that.

   ```
   Finish[i] ==false &&
   Need[i]<= Work
   ```

   If no such ‘i’ is present then go to step 4 else to step 3.

3. The process 'i' selected in the above step runs and finishes its execution. Also, the resources allocated to it gets free. The resources which get free are added to the Work and Finish(i) of the process is set as true. The following operations are performed:

   ```
   Work = Work + Allocation
   Finish[i] = true
   ```
   After performing the 3rd step go to step 2.

4. If all the processes are executed in some sequence then it is said to be a safe state. Or, we can say that if

   ```
   Finish[i]==true for all i,
   ```
   then the system is said to be in a safe state.

Suppose we have 3 processes(A, B, C) and 3 resource types(R1, R2, R3) each having 5 instances. Suppose at any time t if the snapshot of the system taken is as follows then find the system is in a safe state or not.

Total instances of each resources		
R1	R2	R3
5	5	5

Process	Allocation			Max		
	R1	R2	R3	R1	R2	R3
A	1	2	1	2	2	4
B	2	0	1	2	1	3
C	2	2	1	3	4	1
Total_alloc	5	4	3			

Available = Total - Total_alloc
= [5, 5, 5] - [5, 4, 3]
= [0, 1, 2]

So, the total allocated resources(total_alloc)are [5, 4, 3]. Therefore, the Available(the resources that are currently available)resources are

```
Available = [0, 1, 2]
```
Now, we will make the Need Matrix for the system according to the given conditions. As we know, Need(i)=Max(i)-Allocation(i), so the resultant Need matrix will be as follows:

Process	Need		
	R1	R2	R3
A	1	0	3
B	0	1	2
C	1	2	0

Now, we will apply the safety algorithm to check that if the given state is a safe state or not.

1. Work=Available=[0, 1, 2]
2. Also Finish[i]=false, for i=0,1,2, are set as false as none of these processes have been executed.
3. Now, we check that Need[i]≤Work. By seeing the above Need matrix we can tell that only B[0, 1, 2] process can be executed. So, process B(i=1)is allocated the resources and it completes its execution. After completing the execution, it frees up the resources.
4. Again, Work=Work+Available i.e. Work=[0, 1, 2]+[2, 0,1]= [2, 1, 3] and Finish[1]= true.
5. Now, as we have more instances of resources available we will check that if any other process resource needs can be satisfied. With the currently available resources[2, 1, 3], we can see that only process A[1, 2, 1] can be executed. So, process A(i=0) is allocated the resources and it completes its execution. After completing the execution, it frees up the resources.
6. Again, Work=Work+Available i.e. Work=[2, 1, 3]+[1, 2, 1]= [3, 3, 4] and Finish[0]= true.
7. Now, as we have more instances of resources available we will check that if the remaining last process resource requirement can be satisfied. With the currently available resources[3, 3, 4], we can see that process C[2, 2, 1] can be executed. So, process C(i=2) is allocated the resources and it completes its execution. After completing the execution, it frees up the resources.
8. Finally, Work=Work+Available i.e. Work=[3, 3, 4]+[2, 2, 1]= [5, 5, 5] and Finish[2]= true.
9. Finally, all the resources are free and there exists a safe sequence B, A, C in which all the processes can be executed. So. the system is in a safe state and deadlock will not occur.

Dekker's Algorithm for the Critical Section Problem

What is Critical Section Problem ?

Critical Section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.

The critical section cannot be executed by more than one process at the same time; operating system faces the difficulties in allowing and disallowing the processes from entering the critical section.

The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

In order to synchronize the cooperative processes, our main task is to solve the critical section problem. We need to provide a solution in such a way that the following conditions can be satisfied.

The solution to critical section problem must ensure following three conditions:

a. Mutual Exclusion
b. Progress
c. Bounded Waiting

The mutual exclusion problem has different solutions. Dekker was a Dutch mathematician who introduced a software-based solution for the mutual exclusion problem. This algorithm is commonly called Dekker's algorithm. The Deckker's algorithm was developed for an algorithm for mutual exclusion between two processes.

Dekker's algorithm was the first provably-correct solution to the critical section problem. It allows two threads to share a single-use resource without conflict, using only shared memory for communication. It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.

Implementation of Dekker's Algorithm

```
Boolean flag[ 2]
Flag[0]  =  false;
flag[1] =  false;
Int  turn = 1;
Int PID = 0; this value is set for process P1,set this to 1and for process P2.
INT PID = 1 – PID ;
```

In this algorithm, the position of each process is indicated with the variables tern and flag. We can also check which process is in the critical section or not. The variable PID indicates which process is in the critical section. Similarly, we can say here that the PID2 variable represents that which process is not in its critical section.

In the above algorithm, we can see that the value of PID is set as 0 for the first process and we can and say that the first process is in its critical section.

The value of PID is set as 1 for another process that is not in its critical section.

So the value of variable PID and value of variable PID 2 are enough to tell us that a process in its critical section or not.