Banker's Algorithm for Deadlock Avoidance

The OS uses this algorithm each time there is a process request for using a resource that did not used so far. The algorithm *assumes* that it grants the requested resources to the process and it examines whether the resulting system status allows all the process to be completed (it does not lead to a deadlock).

Initially, for each new process entering the system the OS prepares a report with the maximum number of instances of the different resource types, which will be used by the process. For example, the process will use three CPUs (number of CPU instance = 3). Assuming ***m*** different resource types (CPUs, HDs, printers, etc.) and ***n*** processes the system's status can be described by the following data structures:

- ***Available:*** 1-D Array of ***m*** entries***;*** Available(i)=k means that the system has available to use k (it can grant these to the processes) instances of the resource type i.
- ***Max:*** 2-D Array ***m x n*** entries; Max(i,j)=k means that the process j will use up to k instances of the resource type i.
- ***Allocation:*** 2-D Array ***m x n*** entries; Allocation(i,j)=k means that the process j *currently* uses k instances of the resource type i (they were given by the system).
- ***Need:*** 2-D Array ***m x n*** entries; Need(i,j)=k means that the process j, in order to complete its execution, will need (apart the resources that already uses) k instances of the resource type i, which have to be granted by the OS.

For example, consider a system that has labelled the CPUs as i. The system includes 6 CPUs. The process x uses one CPU and process y another one. We have Available(i)=4 because only 2 out of 6 CPUs are currently in use. Process j starts now (entering the system) and the report says that it will use 3 CPUs at most, that is Max(i,j)=3. In the beginning process j uses 1 CPU (requested and granted by the OS) and hence, Allocation(i,j)=1 and the Available will be decreased by 1 CPU: Available(i)=3 and in the future will need 2 more CPUs to complete its computations, hence, Need(i,j)=2. We note here that in all cases Max(i,j)=Allocate(i,j)+Need(i,j).

The banker's Algorithm

The j's process request for additional resources (even when the process j starts the requested resources are considered as additional resources) is reported by a one-dimensional array ***Request*** of ***m*** entries; Request(i)=k denotes that the process j requests k instances of the resource type I to continue its execution. The algorithm examines the following:

- 1. If Request(i) ≤ Need(i,j) then goto 2 else error (for each i requested by process j).
- 2. If Request(i)≤Available(i) then goto 3 else wait (for each i requested by process j).
- 3. If the above checks OK: the algorithm assumes that provides the resources to process j and computes the new status (the data structures):
    - Available(i)=Available(i)-Request(i);
    - Allocation(i,j)=Allocation(i,j)+Request(i);
    - Need(i,j)=Need(i,j)-Request(i);

The algorithm executes the subroutine **Safety** in order to examine if all the processes are able to complete their computations given the new conditions (the resources that were granted to the process j). This is accomplished by the examination of all the processes: if for each process can be completed (Finish) given the available resources (Available). The subroutine **Safety** uses a 1-D array ***n*** Boolean variables, namely the Finish(j). If Finish(j)=true it means that the process j is completed .

Safety

1. For j=1 to n do Finish(j)=false;
2. Find j such that
    a. Finish(j)=false, and
    b. Need(i,j) ≤ Available(i) for all i.

    If there is no such j then goto 4. *// Goto 4 because: either there is no process that can // finish or they have all Finish(j)=true*
3. Execute:
    a. Available(i)=Available(i)+Allocation(i,j);
    b. Finish(j)=true;
    c. Goto step 2.
4. If Finish(j)=true for all j then the system is in a SAFE state.

The subroutine S*afety* requires in the worst case the completion of a loop (step 2) to locate a process **j** that can be completed; it then removes this process from the list of ready to run processes (Finish(j)=true). For this loop in the worst case will check all processes and it will remove the n^{th} (the last one that it checked) thus completing n steps. In the next iteration it may remove the $(n-1)^{th}$ and thus perform n-1 steps, and so on. Therefore, the complexity of the S*afety* is $\mathbf{n^2}$ steps.