BANKER'S ALGORITHM

- Project Name : BANKER'S ALGORITHM
- Language : C++
- Compiler : Borland C Compiler
- Subject : DeadLock Avoidance (*Operating System Concepts*)
- Algorithms
  1) Banker's Algorithm
  2) Deadlock Safety Algorithm
  3) Resource-Request Algorithm
- Files
  1) class.cpp (Banker Class Declaration)
  2) define.cpp (Banker Class Definitions)
  3) graphic.cpp (Graphic Functions)
  4) banker.cpp (Main Program)
- Size : 17.6 KB (18,098 bytes)

Banker's Algorithm

Banker's Algorithm, which is a deadlock avoidance algorithm. It is called the Banker's Algorithm, because it could be used by a bank to make sure that money is allocated in such a way that all customer needs are met. When a new process enters the system, it declares the maximum number of instances that are needed. This number cannot exceed the total number of resources in the system. If the process can be accommodated based upon the needs of the system, then resources are allocated, otherwise the process must wait. The algorithm is actually made up of two separate algorithms: the safety algorithm and the resource allocation algorithm.

The following data structures are needed:

no_of_process represents the number of processes and ***no_of_resource*** represents the number of resource types.

1) *Available*
    1. A vector (array) of available resources of each type
    2. If available[j] = k, then k instances of R_j are available.
2) *Max*
    1. An *no_of_process* by *no_of_resource* matrix
    2. Defines maximum demand for each process
    3. maximum[i][j] = k, then process P_i may request at most k instances of resource R_j.
3) *Allocation*
    1. An *no_of_process* by *no_of_resource* matrix
    2. Defines number of resources of each type currently allocated to each process
    3. allocation[i][j]=k, then process P_i is currently allocated k instances of R_j.
4) *Need*
    1. An *no_of_process* by *no_of_resource* matrix
    2. Indicates remaining resource need of each process
    3. If need[i][j] = k, then process P_i needs k more instances of R_j.
    4. need[i][j] = maximum[i][j] - allocation[i][j]

Safety Algorithm

The safety algorithm is used to determine if a system is in a safe state. It works as follows:

1) *Initialize Work and Finish*
   a. Vectors of length *no_of_resource* and *no_of_resource* respectively
   b. Initialize work = available
   c. finish[i] = false for i = 0, 1, 9, *no_of_process*-1
2) *Find such process which can be allocated resources*

   Find an index i such that both

   i. finish[i] = false

   ii. $need_i \leq work$

   If no such i exists, go to step 4.
3) *Allocate resources*
   a. work = work + $allocation_i$
   b. finish[i] = true.

   Go to step 2.
4) *Check for SafeState*

   If all finish[i] = true

   then the system is in SafeState

   else the system is in UnSafeState

Resource-Request Algorithm

The resource-request algorithm is used to determine whether requests can be safely granted.
It works as follows:

- Let $request_i$ be the request vector for process P_i.
  If $request_i[j] = k$, then process Pi wants k instances of resources Rj.

- When a request is made by process P_i, the following actions are taken:
  1. If $request_i \leq need$, go to step 2;
     otherwise, raise an error condition.
  2. If $request_i \leq available$, go to step 3;
     otherwise P_i must wait since resources are not yet available.
  3. Have system pretend to allocate the requested resources to process P_i by modifying the state as follows:
     $available = available - request_i$
     $allocation_i = allocation_i + request_i$

- If the resulting resource allocation state is safe,
  then transaction is complete and P_i is allocated its resources.

  If unsafe, P_i must wait and old state is restored.