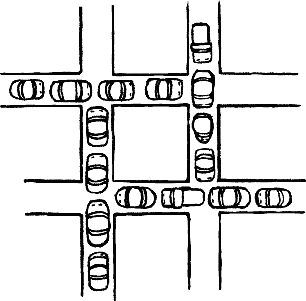
**Process Synchronization By Banker’s Algorithm**

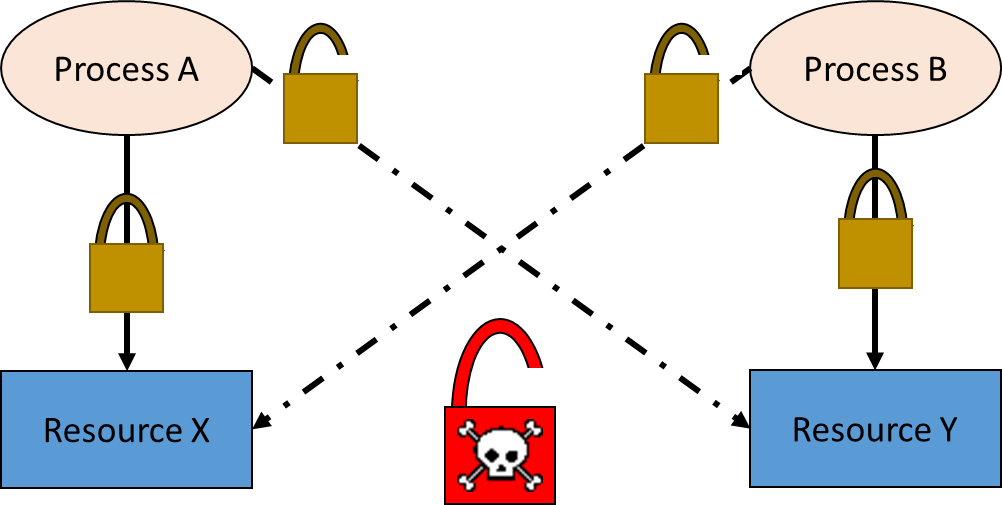


|  |
| --- |
| Name :Ch.Chandra Ram Mohan  Reg : 11611848  Sec : K1620  Roll :B39  Email :mohan.chandaluri22@gmail.com  Link : <https://github.com/mohanchandaluri/Ram/blob/master/bankersalgorithmn.cpp>  <https://github.com/mohanchandaluri/Ram/blob/master/priority%20scheduling.cpp> |

* The bankers algorithm, sometimes referred to as deadlock avoidance algorithm ,is a resource allocation and deadlock avoidance algorithm developed by Edger-Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amount of all resources.
* And then makes an “s-state” check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

Deadlock avoidance algorithm:

* Resource –Allocation Graph is not applicable to a resource allocation system with multiple instance of resource type.
* Supports Multiple instances.
* Named because this algorithm could be used in a bank.



Assumption:

* Each process must a prior claim for maximum use
* When a process requests a resource it may have to wait
* When a process gets all its resources it must return them in a finite amount of time.

Two sub-algorithms:

* Safety algorithm
* Resource-Request algorithm

Data Structures for the Banker’s Algorithm:

let n=number of process and m=number of resources types.

* **Available** : Vector of the length m,if available [j]=k,there are k instances of resource type Rj  available.
* **Max** : n x m matrix. if Max[i,j] then process p1,may request at most k instances type Rj.
* **Allocation** : n x m matrix . if allocation[i,j]=k then Pi is currently allocated k instances of Rj,
* **Need** : n x m matrix If need[i,j]=k,then Pi may need k more instances of Rj to complete its task.

**Need[i,j]=Max[i,j]-Allocation[i,j]**

**Test cases :**

Total system resources are:

A B H I

6 5 7 6

Available system resources are:

A B H I

3 1 1 2

Processes (currently allocated resources):

A B H I

P1 1 2 2 1

P2 1 0 3 3

P3 1 2 1 0

Processes (maximum resources):

A B H I

P1 3 3 2 2

P2 1 2 3 4

P3 1 3 5 0

Need = maximum resources - currently allocated resources

Processes (possibly needed resources):

A B H I

P1 2 1 0 1

P2 0 2 0 1

P3 0 1 4 0

**Safe and Unsafe conditions:**

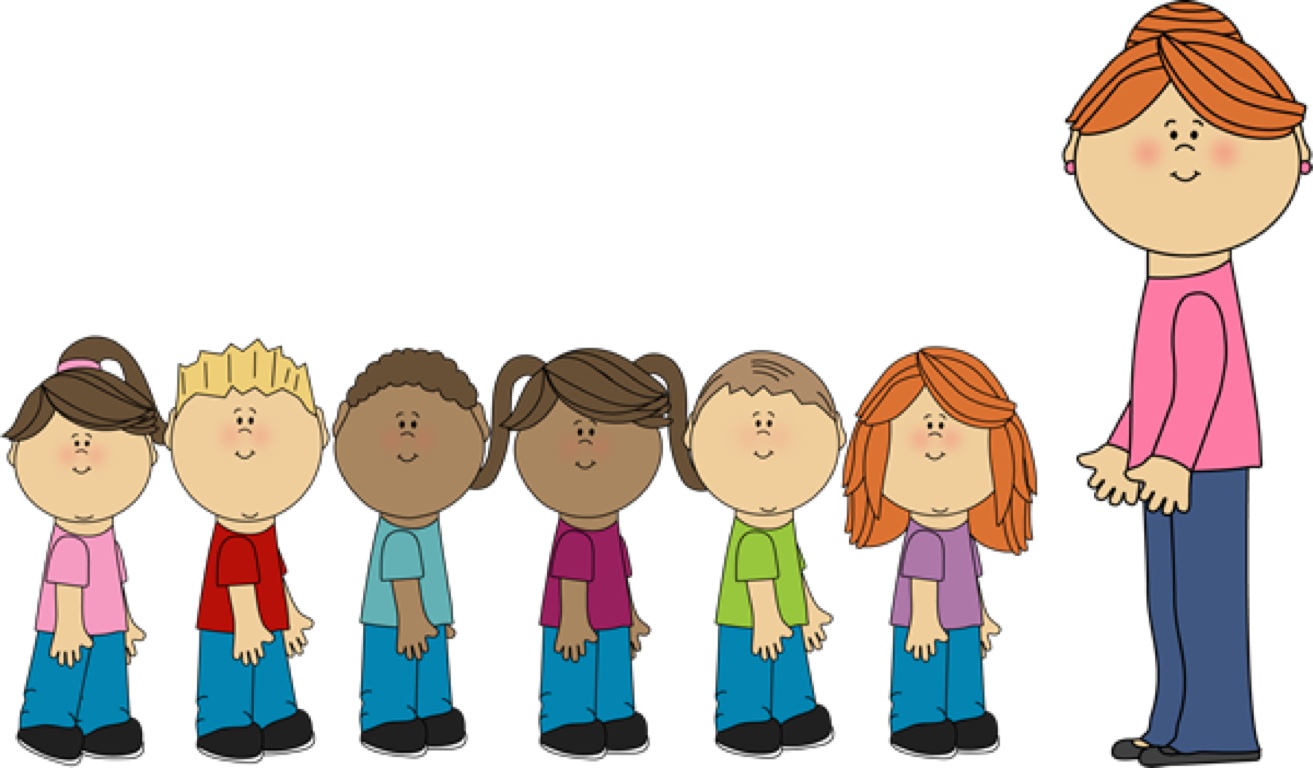
1. P1 acquires 2 A, 1 B and 1 D more resources, achieving its maximum
   * [available resource: <3 1 1 2> - <2 1 0 1> = <1 0 1 1>]
   * The system now still has 1 A, no B, 1 C and 1 D resource available
2. P1 terminates, returning 3 A, 3 B, 2 C and 2 D resources to the system
   * [available resource: <1 0 1 1> + <3 3 2 2> = <4 3 3 3>]
   * The system now has 4 A, 3 B, 3 C and 3 D resources available
3. P2 acquires 2 B and 1 D extra resources, then terminates, returning all its resources
   * [available resource: <4 3 3 3> - <0 2 0 1> + <1 2 3 4> = <5 3 6 6>]
   * The system now has 5 A, 3 B, 6 C and 6 D resources
4. P3 acquires 1 B and 4 C resources and terminates.
   * [available resource: <5 3 6 6> - <0 1 4 0> + <1 3 5 0> = <6 5 7 6>]
   * The system now has all resources: 6 A, 5 B, 7 C and 6 D
5. Because all processes were able to terminate, this state is safe

**Limitations:**

Like the other algorithms, the Banker's algorithm has some limitations when implemented. Specifically, it needs to know how much of each resource a process could possibly request. In most systems, this information is unavailable, making it impossible to implement the Banker's algorithm. Also, it is unrealistic to assume that the number of processes is static since in most systems the number of processes varies dynamically. Moreover, the requirement that a process will eventually release all its resources (when the process terminates) is sufficient for the correctness of the algorithm, however it is not sufficient for a practical system. Waiting for hours (or even days) for resources to be released is usually not acceptable.

\*\*\*-----------------------------------------------------------------------------------------------------\*\*\*

**Process completion by Priority Scheduling**



Priority scheduling is a method of scheduling processes based on priority. In this method, the scheduler chooses the tasks to work as per the priority, which is different from other types of scheduling, for example, a simple round robin.

Priority scheduling involves priority assignment to every process, and processes with higher priorities are carried out first, whereas tasks with equal priorities are carried out on a first-come-first-served (FCFS) or round robin basis. An example of a general-priority-scheduling algorithm is the shortest-job-first (SJF) algorithm.

**Explanation:**

Priorities can be either dynamic or static. Static priorities are allocated during creation, whereas dynamic priorities are assigned depending on the behavior of the processes while in the system. To illustrate, the scheduler could favor input/output (I/O) intensive tasks, which lets expensive requests to be issued as soon as possible.

Priorities may be defined internally or externally. Internally defined priorities make use of some measurable quantity to calculate the priority of a given process. In contrast, external priorities are defined using criteria beyond the operating system (OS), which can include the significance of the process, the type as well as the sum of resources being utilized for computer use, user preference, commerce and other factors like politics, etc.

Priority scheduling can be either of the following:

* Preemptive: This type of scheduling may preempt the central processing unit (CPU) in the case the priority of the freshly arrived process being greater than those of the existing processes.
* Non-preemptive: This type of scheduling algorithm simply places the new process at the top of the ready queue.

**Algorithm:**

**while(j<=max)**

**{i=1;**

**while(i<=n)**

**{**

**if(P[i]==j)**

**{**

**Wt[i]=w;**

**w=w+B[i];**

**}i++;}**

**j++;**

**Test cases :**

Order in which processes gets executed

1 3 2

Processes Burst time Waiting time Turn around time

1 10 0 10

3 8 10 18

2 5 18 23

Average waiting time = 9.33333

Average turn around time = 17

**Disadvantages of Priority Scheduling**:

* Indefinite blocking or starvation.
* A **priority scheduling** can leave some low **priority** waiting processes indefinitely for CPU.
* If the system eventually crashes then all unfinished low **priority** processes gets lost.