**Banker’s Algorithm**

The Banker’s algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of pre-determined maximum possible amounts of all resources, and then makes a ”safe-state” check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue. The Banker’s algorithm is run by the operating system whenever a process requests resources. The algorithm prevents deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

**Resources**

For the Banker’s algorithm to work, it needs to know three things:

1. How much of each resource each process could possibly request

2. How much of each resource each process is currently holding

3. How much of each resource the system has available

Some of the resources that are tracked in real systems are memory, semaphores and interface access.

**Safe and Unsafe States**

A state is considered safe if it is possible for all processes to finish executing (terminate). Since the system cannot know when a process will terminate, or how many resources it will have requested by then, the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward. This is a reasonable assumption in most cases since the system is not particularly concerned with how long each process runs (at least not from a deadlock avoidance perspective). Also, if a process terminates without acquiring its maximum resources, it only makes it easier on the system. Given that assumption, the algorithm determines if a state is safe by trying to find a hypothetical set of requests by the processes that would allow each to acquire (one-by-one) its maximum resources and then terminate (returning its resources to the system).

Any state where no such set exists is an unsafe state. In a safe state, at least one process should be able to acquire its maximum possible set of resources, and proceed to termination.

**The simplified algorithm**

When the system receives a request for resources, it runs the Banker’s algorithm to determine if it is safe to grant the request. The algorithm is fairly straight forward once the distinction between safe and unsafe states is understood.

1. Can the request be granted?

• If not, the request is impossible and must either be denied or put on a waiting list

2. Assume that the request is granted

3. Is the new state safe?

• If so grant the request

• If not, either deny the request or put it on a waiting list

Whether the system denies or postpones an impossible or unsafe request is a decision specific to the operating system

**An Example**

**Introduction:**

Assume we have nine tape drives. Consider whether or not the following states are *safe* or *unsafe*.

|  |  |  |
| --- | --- | --- |
| **State** | **Current Loan** | **Maximum Need** |
| Process A | 0 | 3 |
| Process B | 3 | 5 |
| Process C | 4 | 7 |
|  |  |  |

* Since only 7 (3+4) tape drives are currently on loan (allocated), two (2) tape drives are still available.
* Process B can finish with only two additional tape drives.
* Once Process B is done, it will release all 5 tape drives, making the number of available tape drives = 5.
* With only three of these tape drives, either Process A or Process C may complete and release its tape drives.
* This means that there are two possible ***safe sequences***: <Process B, Process A, Process C> and <Process B, Process C, Process A>.
* Thus, we say that this is a **safe** state.

Again assume we have nine tape drives. Consider whether or not the following states are *safe* or *unsafe*.

|  |  |  |
| --- | --- | --- |
| **State** | **Current Loan** | **Maximum Need** |
| Process A | 5 | 7 |
| Process B | 2 | 5 |
| Process C | 1 | 3 |
|  |  |  |

* Since 8 (5+2+1) tape drives are currently on loan (allocated), only one tape drive is still available.
* None of the three processes can complete with only one additional tape drive.
* This means that there are **no *safe sequences*** possible.
* Thus, we say that this is an **unsafe** state.

Now return to the first example.

Suppose that Process C requests one tape drive.

If this request is granted, will we still be in a safe state?

|  |  |  |
| --- | --- | --- |
| **State** | **Current Loan** | **Maximum Need** |
| Process A | 0 | 3 |
| Process B | 3 | 5 |
| Process C | 5 | 7 |
|  |  |  |

* The number of available tape drives is reduced to one (1).
* No process can be granted enough tape drives to complete.
* This means that there will be **no *safe sequences*** possible, if we grant Process C's request.
* Thus, granting this request will take us from a **safe** state to an **unsafe** state.

**According to Deitel:**

"An unsafe state does not imply the existence of deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to deadlock."

**The Banker's algorithm:**

Allows:

* mutual exclusion
* wait and hold
* no preemption

Prevents:

* circular wait

User process may only request one resource at a time.

System grants request only if the request will result in a safe state.

**The Banker's algorithm: An Example**

Assume we have the following resources:

* 5 tape drives
* 2 graphic displays
* 4 printers
* 3 disks

We can create a vector representing our total resources: **Total** = (5, 2, 4, 3).

Consider we have already allocated these resources among four processes as demonstrated by the following matrix named **Allocation**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Process Name** | **Tape Drives** | **Graphics** | **Printers** | **Disk Drives** |
| Process A | 2 | 0 | 1 | 1 |
| Process B | 0 | 1 | 0 | 0 |
| Process C | 1 | 0 | 1 | 1 |
| Process D | 1 | 1 | 0 | 1 |
|  |  |  |  |  |

The vector representing the allocated resources is the sum of these columns:

**Allocated** = (4, 2, 2, 3).

We also need a matrix to show the number of each resource still needed for each process; we call this matrix **Need**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Process Name** | **Tape Drives** | **Graphics** | **Printers** | **Disk Drives** |
| Process A | 1 | 1 | 0 | 0 |
| Process B | 0 | 1 | 1 | 2 |
| Process C | 3 | 1 | 0 | 0 |
| Process D | 0 | 0 | 1 | 0 |
|  |  |  |  |  |

The vector representing the available resources will be the sum of these columns subtracted from the **Allocated** vector: **Available** = (1, 0, 2, 0).

**The Banker's algorithm:**

1. Find a row in the **Need** matrix which is less than the **Available** vector. If such a row exists, then the process represented by that row may complete with those additional resources. If no such row exists, eventual deadlock is possible.
2. You want to double check that granting these resources to the process for the chosen row will result in a safe state. Looking ahead, pretend that that process has acquired all its needed resources, executed, terminated, and returned resources to the **Available** vector. Now the value of the **Available** vector should be greater than or equal to the value it was previously.
3. Repeat steps 1 and 2 until
   1. all the processes have successfully reached pretended termination (this implies that the initial state was safe); or
   2. deadlock is reached (this implies the initial state was unsafe).

Following the algorithm sketched above,

* Iteration 1:
  1. Examine the **Need** matrix. The only row that is less than the **Available** vector is the one for Process D.

**Need**(Process D) = (0, 0, 1, 0) < (1, 0, 2, 0) = **Available**

* 1. If we assume that Process D completes, it will turn over its currently allocated resources, incrementing the **Available** vector.

|  |  |
| --- | --- |
| (1, 0, 2, 0) | Current value of **Available** |
| +   (1, 1, 0, 1) | **Allocation** (Process D) |
| ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` |  |
| (2, 1, 2, 1) | Updated value of **Available** |
|  |  |

* Iteration 2:
  1. Examine the **Need** matrix, ignoring the row for Process D. The only row that is less than the **Available** vector is the one for Process A.

**Need**(Process A) = (1, 1, 0, 0) < (2, 1, 2, 1) = **Available**

* 1. If we assume that Process A completes, it will turn over its currently allocated resources, incrementing the **Available** vector.

|  |  |
| --- | --- |
| (2, 1, 2, 1) | Current value of **Available** |
| +   (2, 0, 1, 1) | **Allocation** (Process A) |
| ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` |  |
| (4, 1, 3, 2) | Updated value of **Available** |
|  |  |

* Iteration 3:
  1. Examine the **Need** matrix without the row for Process D and Process A. The only row that is less than the **Available** vector is the one for Process B.

**Need**(Process B) = (0, 1, 1, 2) < (4, 1, 3, 2) = **Available**

* 1. If we assume that Process B completes, it will turn over its currently allocated resources, incrementing the **Available** vector.

|  |  |
| --- | --- |
| (4, 1, 3, 2) | Current value of **Available** |
| +   (0, 1, 0, 0) | **Allocation** (Process B) |
| ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` |  |
| (4, 2, 3, 2) | Updated value of **Available** |
|  |  |

* Iteration 4:
  1. Examine the **Need** matrix without the rows for Process A, Process B, and Process D. The only row left is the one for Process C, and it is less than the **Available** vector.

**Need**(Process C) = (3, 1, 0, 0) < (4, 2, 3, 2) = **Available**

* 1. If we assume that Process C completes, it will turn over its currently allocated resources, incrementing the **Available** vector.

|  |  |
| --- | --- |
| (4, 2, 3, 3) | Current value of **Available** |
| +   (1, 0, 1, 1) | **Allocation** (Process C) |
| ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` |  |
| (5, 2, 4, 3) | Updated value of **Available** |
|  |  |

Notice that the final value of the **Available** vector is the same as the original **Total** vector, showing the total number of all resources:

**Total** = (5, 2, 4, 2) < (5, 2, 4, 2) = **Available**

This means that the initial state represented by the **Allocation** and **Need** matrices is a **safe** state.

The safe sequence that assures this safe state is **<D, A, B, C>**.

**Note**: The Banker's algorithm can also be used in the detection of deadlock.

**Disadvantages of the Banker's Algorithm**

* It requires the number of processes to be fixed; no additional processes can start while it is executing.
* It requires that the number of resources remain fixed; no resource may go down for any reason without the possibility of deadlock occurring.
* It allows all requests to be granted in finite time, but one year is a finite amount of time.
* Similarly, all of the processes guarantee that the resources loaned to them will be repaid in a finite amount of time. While this prevents absolute starvation, some pretty hungry processes might develop.
* All processes must know and state their maximum resource need in advance.