Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Català
Deutsch
Español
فارسی
Français
Հայերեն
Italiano
עברית
Magyar
Монгол
日本語
Polski
Português
Slovenčina
Српски / srpski
Українська
中文
Edit links

Banker's algorithm

From Wikipedia, the free encyclopedia

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 predetermined maximum possible amounts 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.

The algorithm was developed in the design process for the THE operating system and originally described (in Dutch) in EWD108.[1] When a new process enters a system, it must declare the maximum number of instances of each resource type that it may ever claim; clearly, that number may not exceed the total number of resources in the system. Also, when a process gets all its requested resources it must return them in a finite amount of time.

Contents [hide]

Resources [edit]

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

- How much of each resource each process could possibly request[MAX]
- How much of each resource each process is currently holding[ALLOCATED]
- How much of each resource the system currently has available[AVAILABLE]

Resources may be allocated to a process only if it satisfies the following conditions:

1. request ≤ max, else set error condition as process has crossed maximum claim made by it.
2. request ≤ available, else process waits until resources are available.

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

The Banker's Algorithm derives its name from the fact that this algorithm could be used in a banking system to ensure that the bank does not run out of resources, because the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. By using the Banker's algorithm, the bank ensures that when customers request money the bank never leaves a safe state. If the customer's request does not cause the bank to leave a safe state, the cash will be allocated, otherwise the customer must wait until some other customer deposits enough.

Basic data structures to be maintained to implement the Banker's Algorithm:

Let n be the number of processes in the system and m be the number of resource types. Then we need the following data structures:

- Available: A vector of length m indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type R_j available.
- Max: An $n×m$ matrix defines the maximum demand of each process. If Max[i,j] = k, then P_i may request at most k instances of resource type R_j.
- Allocation: An $n×m$ matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process P_i is currently allocated k instances of resource type R_j.
- Need: An $n×m$ matrix indicates the remaining resource need of each process. If Need[i,j] = k, then P_i may need k more instances of resource type R_j to complete the task.

Note: Need[i,j] = Max[i,j] - Allocation[i,j].

Example [edit]

```
Available system resources are:
A B C D
3 3 2 1
```

```
Processes (currently allocated resources):
   A B C D
P1 2 0 0 1
P2 3 1 0 1
P3 2 1 1 3
P4 1 3 1 2
P5 1 1 3 2
P6 0 1 0 3
```

```
Processes (maximum resources):
   A B C D
P1 2 4 1 2
P2 5 2 8 2
P3 1 3 1 6
P4 1 4 2 4
P5 3 6 7 5
P6 3 4 5 3
```

```
Need= maximum resources - currently allocated resources
Processes (need resources):
   A B C D
P1 0 4 1 1
P2 2 1 8 1
P3 0 2 0 3
P4 0 1 1 2
P5 2 5 4 3
P6 3 3 5 0
```

Safe and Unsafe States [edit]

A state (as in the above example) 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 resource it only makes it easier on the system. A safe state is considered to be the decision maker if it is going to process ready queue. Safe State ensures the Security.

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 its maximum resources and then terminate (returning its resources to the system). Any state where no such set exists is an **unsafe** state.

We can show that the state given in the previous example is a safe state by showing that it is possible for each process to acquire its maximum resources and then terminate.

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

For an example of an unsafe state, consider what would happen if process 2 was holding 1 more unit of resource B at the beginning.

Requests [edit]

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.

Example [edit]

Starting in the same state as the previous example started in, assume process 3 requests 2 units of resource C.

1. There is not enough of resource C available to grant the request
2. The request is denied

On the other hand, assume process 3 requests 1 unit of resource C.

1. There are enough resources to grant the request
2. Assume the request is granted
   - The new state of the system would be:

```
    Available system resources
     A B C D
Free 3 1 0 2
```

```
    Processes (currently allocated resources):
     A B C D
P1   1 2 2 1
P2   1 0 3 3
P3   1 2 2 0
```

```
    Processes (maximum resources):
     A B C D
P1   3 3 2 2
P2   1 2 3 4
P3   1 3 5 0
```

1. Determine if this new state is safe
   1. P1 can acquire 2 A, 1 B and 1 D resources and terminate
   2. Then, P2 can acquire 2 B and 1 D resources and terminate
   3. Finally, P3 can acquire 1B and 3 C resources and terminate
   4. Therefore, this new state is safe
2. Since the new state is safe, grant the request

Finally, from the state we started at, assume that process 2 requests 1 unit of resource B.

1. There are enough resources
2. Assuming the request is granted, the new state would be:

```
    Available system resources:
     A B C D
Free 3 0 1 2
```

```
    Processes (currently allocated resources):
     A B C D
P1   1 2 2 1
P2   1 1 3 3
P3   1 2 1 0
```

```
    Processes (maximum resources):
     A B C D
P1   3 3 2 2
P2   1 2 3 4
P3   1 3 5 0
```

1. Is this state safe? Assuming P1, P2, and P3 request more of resource B and C.
   - P1 is unable to acquire enough B resources
   - P2 is unable to acquire enough B resources
   - P3 is unable to acquire enough B resources
   - No process can acquire enough resources to terminate, so this state is not safe
2. Since the state is unsafe, deny the request

```
/*PROGRAM TO IMPLEMENT BANKER'S ALGORITHM
  *   --------------------------------------------*/
#include <stdio.h>
int curr[5][5], maxclaim[5][5], avl[5];
int alloc[5] = {0, 0, 0, 0, 0};
int maxres[5], running[5], safe=0;
int count = 0, i, j, exec, r, p, k = 1;

int main()
{
    printf("\nEnter the number of processes: ");
    scanf("%d", &p);

    for (i = 0; i < p; i++) {
        running[i] = 1;
        count++;
    }

    printf("\nEnter the number of resources: ");
    scanf("%d", &r);

    printf("\nEnter Claim Vector:");
    for (i = 0; i < r; i++) {
        scanf("%d", &maxres[i]);
    }

    printf("\nEnter Allocated Resource Table:\n");
    for (i = 0; i < p; i++) {
        for(j = 0; j < r; j++) {
            scanf("%d", &curr[i][j]);
        }
    }

    printf("\nEnter Maximum Claim Table:\n");
    for (i = 0; i < p; i++) {
        for(j = 0; j < r; j++) {
            scanf("%d", &maxclaim[i][j]);
        }
    }

    printf("\nThe Claim Vector is: ");
    for (i = 0; i < r; i++) {
        printf("\t%d", maxres[i]);
    }

    printf("\nThe Allocated Resource Table:\n");
    for (i = 0; i < p; i++) {
        for (j = 0; j < r; j++) {
            printf("\t%d", curr[i][j]);
        }

        printf("\n");
    }

    printf("\nThe Maximum Claim Table:\n");
    for (i = 0; i < p; i++) {
        for (j = 0; j < r; j++) {
            printf("\t%d", maxclaim[i][j]);
        }

        printf("\n");
    }

    for (i = 0; i < p; i++) {
        for (j = 0; j < r; j++) {
            alloc[j] += curr[i][j];
        }
    }

    printf("\nAllocated resources:");
    for (i = 0; i < r; i++) {
        printf("\t%d", alloc[i]);
    }

    for (i = 0; i < r; i++) {
        avl[i] = maxres[i] - alloc[i];
    }

    printf("\nAvailable resources:");
    for (i = 0; i < r; i++) {
        printf("\t%d", avl[i]);
    }
    printf("\n");

    //Main procedure goes below to check for unsafe state.
    while (count != 0) {
        safe = 0;
        for (i = 0; i < p; i++) {
            if (running[i]) {
                exec = 1;
                for (j = 0; j < r; j++) {
                    if (maxclaim[i][j] - curr[i][j] > avl[j]) {
                        exec = 0;
                        break;
                    }
                }
                if (exec) {
                    printf("\nProcess%d is executing\n", i + 1);
                    running[i] = 0;
                    count--;
                    safe = 1;

                    for (j = 0; j < r; j++) {
                        avl[j] += curr[i][j];
                    }

                    break;
                }
            }
        }
        if (!safe) {
            printf("\nThe processes are in unsafe state.\n");
            break;
        } else {
            printf("\nThe process is in safe state");
            printf("\nAvailable vector:");

            for (i = 0; i < r; i++) {
                printf("\t%d", avl[i]);
            }

            printf("\n");
        }
    }
}

/*SAMPLE  OUTPUT
-----------------
Enter the number of processes:5

Enter the number of resources:4

Enter Claim Vector:8 5 9 7

Enter Allocated Resource Table:
2 0 1 1
0 1 2 1
4 0 0 3
0 2 1 0
1 0 3 0

Enter Maximum Claim Table:
3 2 1 4
0 2 5 2
5 1 0 5
1 5 3 0
3 0 3 3

The Claim Vector is: 8 5 9 7
The Allocated Resource Table:
 2 0 1 1
 0 1 2 1
 4 0 0 3
 0 2 1 0
 1 0 3 0

The  Maximum Claim Table:
 3 2 1 4
 0 2 5 2
 5 1 0 5
 1 5 3 0
 3 0 3 3

 Allocated resources: 7 3 7 5
 Available resources: 1 2 2 2

Process3 is executing

 The process is in safe state
 Available vector: 5 2 2 5
Process1 is executing

 The process is in safe state
 Available vector: 7 2 3 6
Process2 is executing

 The process is in safe state
 Available vector: 7 3 5 7
Process4 is executing

 The process is in safe state
 Available vector: 7 5 6 7
```

```
Process5 is executing

 The process is in safe state
 Available vector: 8 5 9 7

 -------------------------------------------------------*/
```

Limitations [edit]

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.

References [edit]

1. ^ Dijkstra, Edsger W. *Een algorithme ter voorkoming van de dodelijke omarming (EWD-108)*. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (original; transcription) (in Dutch; *An algorithm for the prevention of the deadly embrace*)

Further reading [edit]

- "Operating System Concepts" by Silberschatz, Galvin, and Gagne (pages 259-261 of the 7th edition)
- "Operating System Concepts" by Silberschatz, Galvin, and Gagne (pages 298-300 of the 8th edition)
- Dijkstra, Edsger W. *The mathematics behind the Banker's Algorithm (EWD-623)*. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (original; transcription) (1977), published as pages 308–312 of Edsger W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982. ISBN 0-387-90652-5

External links [edit]

- Deadlock Recovery, Avoidance and Prevention

Categories: Concurrency control algorithms | Dutch inventions