Banker's algorithm

From Wikipedia, the free encyclopedia

The **Banker's algorithm** is a resource allocation & 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 algorithm was developed in the design process for the THE operating system and originally described (in Dutch) in EWD108[1]. The name is by analogy with the way that bankers account for liquidity constraints.

Contents

Algorithm

The Banker's algorithm is run by the operating system whenever a process requests resources.[2] 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). When a new process enters a system, it must declare the maximum number of instances of each resource type that may not exceed the total number of resources in the system.

Resources

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

- How much of each resource each process could possibly request
- How much of each resource each process is currently holding
- 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.

Example

Assuming that the system distinguishes between four types of resources, (A, B, C and D), the following is an example of how those resources could be distributed. *Note that this example shows the system at an instant before a new request for resources arrives. Also, the types and number of resources are abstracted. Real systems, for example, would deal with much larger quantities of each resource.*

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

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

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

Safe and Unsafe States

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

Pseudo-Code[3]

P - set of processes

Mp - maximal requirement of resources for process p

Cp - current resources allocation process p

A - currently available resources

```
while (P != ∅) {
    found = False
    foreach (p ∈ P) {
        if (Mp − Cp ≤ A) {
             /* p can obtain all it needs.         */
             /* assume it does so, terminates, and */
             /* releases what it already has.      */
             A = A + Cp
             P = P − {p}
             found = True
        }
    }
    if (!found) return UNSAFE
}
return SAFE
```

Example

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
   - 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
   - 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

    - The system now has 5 A, 3 B, 6 C and 6 D resources

4. P3 acquires 4 C resources and terminates
    - The system now has all resources: 6 A, 4 B, 7 C and 6 D
5. Because all processes were able to terminate, this state is safe

Note that these requests and acquisitions are *hypothetical*. The algorithm generates them to check the safety of the state, but no resources are actually given and no processes actually terminate. Also note that the order in which these requests are generated – if several can be fulfilled – doesn't matter, because all hypothetical requests let a process terminate, thereby increasing the system's free resources.

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

Requests

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

Continuing the previous examples, 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 1 2 0
```

```
    Processes (maximum resources):
     A B C D
P1   3 3 2 2
P2   1 2 3 4
P3   1 1 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 3 C resources and terminate
    4. Therefore, this new state is safe
2. Since the new state is safe, grant the request

Finally, 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 1 1 0
```

```
    Processes (maximum resources):
     A B C D
P1   3 3 2 2
P2   1 2 3 4
P3   1 1 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 C resources
   - No process can acquire enough resources to terminate, so this state is not safe
2. Since the state is unsafe, deny the request

Note that in this example, no process was able to terminate. It is possible that some processes will be able to terminate, but not all of them. That would still be an unsafe state.

Trade-offs

Like most algorithms, the Banker's algorithm involves some trade-offs. Specifically, it needs to know how much of each resource a process could possibly request. In most systems, this information is unavailable, making the Banker's algorithm useless. Besides, it is unrealistic to assume that the number of processes is static. 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

1. ^ E. W. Dijkstra "EWD108: Een algorithme ter voorkoming van de dodelijke omarming (http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF) " (in Dutch; *An algorithm for the prevention of the deadly embrace*)
2. ^ Bic, Lubomir F.; Shaw, Alan C. (2003). *Operating System Principles (http://vig.prenhall.com/catalog/academic/product /0,1144,0130266116,00.html)* . Prentice Hall. ISBN 0-13-026611-6. http://vig.prenhall.com/catalog/academic/product /0,1144,0130266116,00.html.
3. ^ Concurrency (http://www.cs.huji.ac.il/course/2006/os/notes/notes4.pdf)

Further reading

- "Operating System Concepts (http://codex.cs.yale.edu/avi/os-book/os7) " by Silberschatz, Galvin, and Gagne (pages 259-261 of the 7th edition)
- "EWD623: The mathematics behind the Banker’s Algorithm (http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD623.PDF) " (1977) by E.

W. Dijkstra, 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

- Deadlock Recovery, Avoidance and Prevention (http://www.isi.edu/~faber/cs402/notes/lecture9.html)

Retrieved from "http://en.wikipedia.org/wiki/Banker%27s_algorithm"
Categories: Concurrency control algorithms