Create account Log in

Article Talk Read Edit View history Search

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
한국어
Հայերեն
Bahasa Indonesia
Italiano
日本語
Polski
Русский
Српски / srpski
Türkçe
Українська
Edit links

Dekker's algorithm

From Wikipedia, the free encyclopedia

Dekker's algorithm is the first known correct solution to the mutual exclusion problem in concurrent programming. The solution is attributed to Dutch mathematician Th. J. Dekker by Edsger W. Dijkstra in an unpublished paper on sequential process descriptions[1] and his manuscript on cooperating sequential processes.[2] It allows two threads to share a single-use resource without conflict, using only shared memory for communication.

It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.

Contents [hide]

Overview [edit]

If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is. If one process is already in the critical section, the other process will busy wait for the first process to exit. This is done by the use of two flags, `wants_to_enter[0]` and `wants_to_enter[1]`, which indicate an intention to enter the critical section on the part of processes 0 and 1, respectively, and a variable `turn` that indicates who has priority between the two processes.

Dekker's algorithm can be expressed in pseudocode, as follows.[3]

```
    variables
        wants_to_enter : array of 2 booleans
        turn : integer

    wants_to_enter[0] ← false
    wants_to_enter[1] ← false
    turn ← 0   // or 1
```

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ←
false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ←
true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ←
false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ←
true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

Processes indicate an intention to enter the critical section which is tested by the outer while loop. If the other process has not flagged intent, the critical section can be entered safely irrespective of the current turn. Mutual exclusion will still be guaranteed as neither process can become critical before setting their flag (implying at least one process will enter the while loop). This also guarantees progress as waiting will not occur on a process which has withdrawn intent to become critical. Alternatively, if the other process's variable was set the while loop is entered and the turn variable will establish who is permitted to become critical. Processes without priority will withdraw their intention to enter the critical section until they are given priority again (the inner while loop). Processes with priority will break from the while loop and enter their critical section.

Dekker's algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvation. Let us see why the last property holds. Suppose p0 is stuck inside the "while wants_to_enter[1]" loop forever. There is freedom from deadlock, so eventually p1 will proceed to its critical section and set turn = 0 (and the value of turn will remain unchanged as long as p0 doesn't progress). Eventually p0 will break out of the inner "while turn ≠ 0" loop (if it was ever stuck on it). After that it will set wants_to_enter[0] to true and settle down to waiting for wants_to_enter[1] to become false (since turn = 0, it will never do the actions in the while loop). The next time p1 tries to enter its critical section, it will be forced to execute the actions in its "while wants_to_enter[0]" loop. In particular, it will eventually set wants_to_enter[1] to false and get stuck in the "while turn ≠ 1" loop (since turn remains 0). The next time control passes to p0, it will exit the "while wants_to_enter[1]" loop and enter its critical section.

If the algorithm were modified by performing the actions in the "while wants_to_enter[1]" loop without checking if turn = 0, then there is a possibility of starvation. Thus all the steps in the algorithm are necessary.

Note [edit]

This section **does not cite any references or sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(May 2015)*

One advantage of this algorithm is that it doesn't require special Test-and-set (atomic read/modify/write) instructions and is therefore highly portable between languages and machine architectures. One disadvantage is that it is limited to two processes and makes use of busy waiting instead of process suspension. (The use of busy waiting suggests that processes should spend a minimum of time inside the critical section.)

Modern operating systems provide mutual exclusion primitives that are more general and flexible than Dekker's algorithm. However, in the absence of actual contention between the two processes, the entry and exit from critical section is extremely efficient when Dekker's algorithm is used.

Many modern CPUs execute their instructions in an out-of-order fashion; even memory accesses can be reordered (see memory ordering). This algorithm won't work on SMP machines equipped with these CPUs without the use of memory barriers.

Additionally, many optimizing compilers can perform transformations that will cause this algorithm to fail regardless of the platform. In many languages, it is legal for a compiler to detect that the flag variables *wants_to_enter[0]* and *wants_to_enter[1]* are never accessed in the loop. It can then remove the writes to those variables from the loop, using a process called Loop-invariant code motion. It would also be possible for many compilers to detect that the *turn* variable is never modified by the inner loop, and perform a similar transformation, resulting in a potential infinite loop. If either of these transformations is performed, the algorithm will fail, regardless of architecture.

To alleviate this problem, volatile variables should be marked as modifiable outside the scope of the currently executing context. For example, in C# or Java, one would annotate these variables as 'volatile'. Note however that the C/C++ "volatile" attribute only guarantees that the compiler generates code with the proper ordering; it does not include the necessary memory barriers to guarantee in-order *execution* of that code. C++11 atomic variables can be used to guarantee the appropriate ordering requirements — by default, operations on atomic variables are sequentially consistent so if the wants_to_enter and turn variables are atomic a naive implementation will "just work". Alternatively, ordering can be guaranteed by the explicit use of separate fences, with the load and store operations using a relaxed ordering.

See also [edit]

- Eisenberg & McGuire algorithm
- Peterson's algorithm
- Lamport's bakery algorithm
- Szymanski's algorithm

- Semaphores

References [edit]

1. ^ Dijkstra, Edsger W. *Over de sequentialiteit van procesbeschrijvingen (EWD-35)*. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (original; transcription) (undated, 1962 or 1963); English translation About the sequentiality of process descriptions
2. ^ Dijkstra, Edsger W. *Cooperating sequential processes (EWD-123)*. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (original; transcription) (September 1965)
3. ^ Alagarsamy, K. (2003). "Some Myths About Famous Mutual Exclusion Algorithms". *ACM SIGACT News* **34** (3): 94–103.

Categories: Concurrency control algorithms | Dutch inventions