Mutual exclusion

From Wikipedia, the free encyclopedia

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
العربية
Català
Čeština
Deutsch
Ελληνικά
Español
Euskara
فارسی
Français
한국어
Italiano
עברית
Nederlands
日本語
Polski
Português
Русский
Српски / srpski
Українська
中文
Edit links

It has been suggested that *Critical section* be merged into this article. (Discuss) *Proposed since May 2015.*

For the concept in logic and probability theory, see Mutual exclusivity.
"Mutex" redirects here. For the synchronization device commonly used to establish mutual exclusion, see lock (computer science).

In computer science, **mutual exclusion** refers to the requirement of ensuring that no two concurrent processes[a] are in their critical section at the same time; it is a basic requirement in concurrency control, to prevent race conditions. Here, a critical section refers to a period when the process accesses a shared resource, such as shared memory. The requirement of mutual exclusion was first identified and solved by Edsger W. Dijkstra in his seminal 1965 paper titled *Solution of a problem in concurrent programming control*,[1][2] and is credited as the first topic in the study of concurrent algorithms.[3]

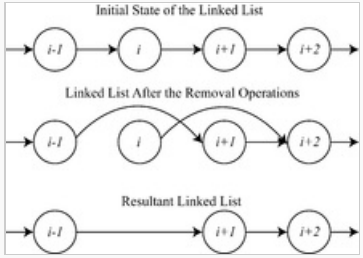

Figure 1: Two nodes, *i* and *i+1*, being removed simultaneously result in node *i+1* not being removed.

A simple example of why mutual exclusion is important in practice can be visualized using a singly linked list (See Figure 1). In such a linked list, the removal of a node is done by changing the "next" pointer of the preceding node to point to the subsequent node (e.g., if node *i* is being removed then the "next" pointer of node *i−1* will be changed to point to node *i+1*). In an execution where such a linked list is being shared between multiple processes, two processes may attempt to remove two different nodes simultaneously, resulting in the following problem: let nodes *i* and *i+1* be the nodes to be removed; furthermore, let neither of them be the head nor the tail; the next pointer of node *i−1* will be changed to point to node *i+1* and the next pointer of node *i* will be changed to point to node *i+2*. Although both removal operations complete successfully, node *i+1* remains in the list since *i−1* was made to point to *i+1*, skipping node *i* (which was the node that reflected the removal of *i+1* by having its next pointer set to *i+2*). This can be seen in Figure 1. This problem (normally called a race condition) can be avoided by using the requirement of mutual exclusion to ensure that simultaneous updates to the same part of the list cannot occur.

Contents [hide]

Enforcing mutual exclusion [edit]

There are both software and hardware solutions for enforcing mutual exclusion. Some different solutions are discussed below.

Hardware solutions [edit]

On uniprocessor systems, the simplest solution to achieve mutual exclusion is to disable interrupts during a process's critical section. This will prevent any interrupt service routines from running (effectively preventing a process from being preempted). Although this solution is effective, it leads to many problems. If a critical section is long, then the system clock will drift every time a critical section is executed because the timer interrupt is no longer serviced, so tracking time is impossible during the critical section. Also, if a process halts during its critical section, control will never be returned to another process, effectively halting the entire system. A more

elegant method for achieving mutual exclusion is the busy-wait.

Busy-waiting is effective for both uniprocessor and multiprocessor systems. The use of shared memory and an atomic test-and-set instruction provides the mutual exclusion. A process can test-and-set on a location in shared memory, and since the operation is atomic, only one process can set the flag at a time. Any process that is unsuccessful in setting the flag can either go on to do other tasks and try again later, release the processor to another process and try again later, or continue to loop while checking the flag until it is successful in acquiring it. Preemption is still possible, so this method allows the system to continue to function - even if a process halts while holding the lock.

Several other atomic operations can be used to provide mutual exclusion of data structures; most notable of these is compare-and-swap (CAS). CAS can be used to achieve wait-free mutual exclusion for any shared data structure by creating a linked list where each node represents the desired operation to be performed. CAS is then used to change the pointers in the linked list[4] during the insertion of a new node. Only one process can be successful in its CAS; all other processes attempting to add a node at the same time will have to try again. Each process can then keep a local copy of the data structure, and upon traversing the linked list, can perform each operation from the list on its local copy.

Software solutions [edit]

Beside hardware-supported solutions, some software solutions exist that use busy waiting to achieve mutual exclusion. Examples of these include the following:

- Dekker's algorithm
- Peterson's algorithm
- Lamport's bakery algorithm[5]
- Szymanski's Algorithm
- Taubenfeld's black-white bakery algorithm.[2]

These algorithms do not work if out-of-order execution is used on the platform that executes them. Programmers have to specify strict ordering on the memory operations within a thread.[6]

It is often preferable to use synchronization facilities provided by an operating system's multithreading library, which will take advantage of hardware solutions if possible but will use software solutions if no hardware solutions exist. For example, when the operating system's lock library is used and a thread tries to acquire an already acquired lock, the operating system could suspend the thread using a context switch and swap it out with another thread that is ready to be run, or could put that processor into a low power state if there is no other thread that can be run. Therefore, most modern mutual exclusion methods attempt to reduce latency and busy-waits by using queuing and context switches. However, if the time that is spent suspending a thread and then restoring it can be proven to be always more than the time that must be waited for a thread to become ready to run after being blocked in a particular situation, then spinlocks are an acceptable solution (for that situation only).[*citation needed*]

Types of mutual exclusion devices [edit]

The solutions explained above can be used to build the synchronization primitives below:

- Locks
- Reentrant mutexes
- Semaphores
- Monitors
- Message passing
- Tuple space
- Readers-writer lock

Many forms of mutual exclusion have side-effects. For example, classic semaphores permit deadlocks, in which one process gets a semaphore, another process gets a second semaphore, and then both wait forever for the other semaphore to be released. Other common side-effects include starvation, in which a process never gets sufficient resources to run to completion; priority inversion, in which a higher priority thread waits for a lower-priority thread; and "high latency", in which response to interrupts is not prompt.

Much research is aimed at eliminating the above effects, often with the goal of guaranteeing non-blocking progress. No perfect scheme is known. Blocking system calls used to sleep an entire process. Until such calls became thread safe, there was no proper mechanism for sleeping a single thread within a process (see polling).

See also [edit]

- Atomicity (programming)
- Concurrency control
- Exclusive or
- Mutually exclusive events
- Semaphore
- Dining philosophers problem
- Reentrant mutex
- Spinlock

Notes [edit]

a. ^ These processes may be implemented as threads.

References [edit]

1. ^ Dijkstra, E. W. (1965). "Solution of a problem in concurrent programming control". *Communications of the ACM* **8** (9): 569. doi:10.1145/365559.365617.
2. ^ [a] [b] Taubenfeld, The Black-White Bakery Algorithm. In Proc. Distributed Computing, 18th international conference, DISC 2004. Vol 18, 56-70, 2004
3. ^ "PODC Influential Paper Award: 2002", *ACM Symposium on Principles of Distributed Computing*, retrieved 2009-08-24
4. ^ https://timharris.uk/papers/2001-disc.pdf
5. ^ Lamport, Leslie (August 1974). "A new solution of Dijkstra's concurrent programming problem". *Communications of the ACM* **17** (8): 453–455. doi:10.1145/361082.361093.
6. ^ Holzmann, Gerard J.; Bosnacki, Dragan (1 October 2007). "The Design of a Multicore Extension of the SPIN Model Checker". *IEEE Transactions on Software Engineering* **33** (10): 659–674. doi:10.1109/TSE.2007.70724.

Further reading [edit]

- Michel Raynal: *Algorithms for Mutual Exclusion*, MIT Press, ISBN 0-262-18119-3
- Sunil R. Das, Pradip K. Srimani: *Distributed Mutual Exclusion Algorithms*, IEEE Computer Society, ISBN 0-8186-3380-8
- Thomas W. Christopher, George K. Thiruvathukal: *High-Performance Java Platform Computing*, Prentice Hall, ISBN 0-13-016164-0
- Gadi Taubenfeld, *Synchronization Algorithms and Concurrent Programming*, Pearson/Prentice Hall, ISBN 0-13-197259-6

External links [edit]

- Article A Simple Mutex Program
- Common threads: POSIX threads explained - The little things called mutexes" by Daniel Robbins
- Mutual Exclusion Petri Net
- Mutual Exclusion with Locks - an Introduction
- Mutual exclusion variants in OpenMP
- The Black-White Bakery Algorithm

Categories: Concurrency control