WIKIPEDIA

Spinlock

In software engineering, a **spinlock** is a lock which causes a thread trying to acquire it to simply wait in a loop ("spin") while repeatedly checking if the lock is available. Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting. Once acquired, spinlocks will usually be held until they are explicitly released, although in some implementations they may be automatically released if the thread being waited on (that which holds the lock) blocks, or "goes to sleep".

Because they avoid overhead from operating system process rescheduling or context switching, spinlocks are efficient if threads are likely to be blocked for only short periods. For this reason, operating-system kernels often use spinlocks. However, spinlocks become wasteful if held for longer durations, as they may prevent other threads from running and require rescheduling. The longer a thread holds a lock, the greater the risk that the thread will be interrupted by the OS scheduler while holding the lock. If this happens, other threads will be left "spinning" (repeatedly trying to acquire the lock), while the thread holding the lock is not making progress towards releasing it. The result is an indefinite postponement until the thread holding the lock can finish and release it. This is especially true on a single-processor system, where each waiting thread of the same priority is likely to waste its quantum (allocated time where a thread can run) spinning until the thread that holds the lock is finally finished.

Implementing spin locks correctly offers challenges because programmers must take into account the possibility of simultaneous access to the lock, which could cause race conditions. Generally, such implementation is possible only with special assembly-language instructions, such as atomic test-and-set operations, and cannot be easily implemented in programming languages not supporting truly atomic operations.[1] On architectures without such operations, or if high-level language implementation is required, a non-atomic locking algorithm may be used, e.g. Peterson's algorithm. But note that such an implementation may require more memory than a spinlock, be slower to allow progress after unlocking, and may not be implementable in a high-level language if out-of-order execution is allowed.

Contents

Example implementation

The following example uses x86 assembly language to implement a spinlock. It will work on any Intel 80386 compatible processor.

```
; Intel syntax

locked:                      ; The lock variable. 1 = locked, 0 = unlocked.
     dd      0

spin_lock:
     mov     eax, 1          ; Set the EAX register to 1.

     xchg    eax, [locked]   ; Atomically swap the EAX register with
                             ;  the lock variable.
                             ; This will always store 1 to the lock, leaving
                             ;  the previous value in the EAX register.
```

```
    test    eax, eax        ; Test EAX with itself. Among other things, this will
                            ;  set the processor's Zero Flag if EAX is 0.
                            ; If EAX is 0, then the lock was unlocked and
                            ;  we just locked it.
                            ; Otherwise, EAX is 1 and we didn't acquire the lock.

    jnz     spin_lock       ; Jump back to the MOV instruction if the Zero Flag is
                            ;  not set; the lock was previously locked, and so
                            ; we need to spin until it becomes unlocked.

    ret                     ; The lock has been acquired, return to the calling
                            ;  function.

spin_unlock:
    mov     eax, 0          ; Set the EAX register to 0.

    xchg    eax, [locked]   ; Atomically swap the EAX register with
                            ;  the lock variable.

    ret                     ; The lock has been released.
```

Significant optimizations

The simple implementation above works on all CPUs using the x86 architecture. However, a number of performance optimizations are possible:

On later implementations of the x86 architecture, *spin_unlock* can safely use an unlocked MOV instead of the slower locked XCHG. This is due to subtle memory ordering rules which support this, even though MOV is not a full memory barrier. However, some processors (some Cyrix processors, some revisions of the Intel Pentium Pro (due to bugs), and earlier Pentium and i486 SMP systems) will do the wrong thing and data protected by the lock could be corrupted. On most non-x86 architectures, explicit memory barrier or atomic instructions (as in the example) must be used. On some systems, such as IA-64, there are special "unlock" instructions which provide the needed memory ordering.

To reduce inter-CPU bus traffic, code trying to acquire a lock should loop reading without trying to write anything until it reads a changed value. Because of MESI caching protocols, this causes the cache line for the lock to become "Shared"; then there is remarkably *no* bus traffic while a CPU waits for the lock. This optimization is effective on all CPU architectures that have a cache per CPU, because MESI is so widespread.

Alternatives

The primary disadvantage of a spinlock is that, while waiting to acquire a lock, it wastes time that might be productively spent elsewhere. There are two ways to avoid this:

1. Do not acquire the lock. In many situations it is possible to design data structures that do not require locking e.g. by using per-thread or per-CPU data and disabling interrupts.
2. Switch to a different thread while waiting. This typically involves attaching the current thread to a queue of threads waiting for the lock, followed by switching to another thread that is ready to do some useful work. This scheme also has the advantage that it guarantees that resource starvation does not occur as long as all threads eventually relinquish locks they acquire and scheduling decisions can be made about which thread should progress first. Spinlocks that never entail switching, usable by real-time operating systems, are sometimes called *raw spinlocks*.[2]

Most operating systems (including Solaris, Mac OS X and FreeBSD) use a hybrid approach called "adaptive mutex". The idea is to use a spinlock when trying to access a resource locked by a currently-running thread, but to sleep if the thread is not currently running. (The latter is *always* the case on single-processor systems.)[3]

OpenBSD attempted to replace spinlocks with ticket locks which enforced first-in-first-out behaviour, however this resulted in more CPU usage in the kernel and larger applications, such as Firefox, becoming much slower.[4][5]

See also

- Synchronization

- Busy spin
- Deadlock
- Seqlock
- Ticket lock

References

1. Silberschatz, Abraham; Galvin, Peter B. (1994). *Operating System Concepts* (Fourth ed.). Addison-Wesley. pp. 176–179. ISBN 0-201-59292-4.
2. Jonathan Corbet (9 December 2009). "Spinlock naming resolved" (https://lwn.net/Articles/365863/). LWN.net. Retrieved 14 May 2013.
3. Silberschatz, Abraham; Galvin, Peter B. (1994). *Operating System Concepts* (Fourth ed.). Addison-Wesley. p. 198. ISBN 0-201-59292-4.
4. Ted Unangst (2013-06-01). "src/lib/librthread/rthread.c - Revision 1.71" (http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/lib/librthread/rthread.c?rev=1.71&content-type=text/x-cvsweb-markup)
5. Ted Unangst (2016-05-06). "tedu comment on Locking in WebKit - Lobsters" (https://lobste.rs/c/6cybxn)

External links

- pthread_spin_lock documentation from The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition
- Variety of spinlock Implementations from Concurrency Kit
- Article "User-Level Spin Locks - Threads, Processes & IPC" by Gert Boddaert
- Paper "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors" by Thomas E. Anderson
- Paper "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors" by John M. Mellor-Crummey and Michael L. Scott. This paper received the 2006 Dijkstra Prize in Distributed Computing
- Spin-Wait Lock by Jeffrey Richter
- Austria C++ SpinLock Class Reference
- Interlocked Variable Access(Windows)
- Operating Systems: Three Easy Pieces (Chapter: Locks)