How is thread synchronization implemented, at the assembly language level?

While I'm familiar with concurrent programming concepts such as mutexes and semaphores, I have never understood how they are implemented at the assembly language level.

I imagine there being a set of memory "flags" saying:

- lock A is held by thread 1
- lock B is held by thread 3
- lock C is not held by any thread
- etc

But how is access to these flags synchronized between threads? Something like this naive example would only create a race condition:

```
  mov edx, [myThreadId]
wait:
  cmp [lock], 0
  jne wait
  mov [lock], edx
  ; I wanted an exclusive lock but the above
  ; three instructions are not an atomic operation :(
```

multithreading assembly concurrency synchronization x86

edited Mar 27 '10 at 20:02
Andras Vass
8,276 18 37

asked Mar 3 '10 at 1:13
Martin
14.9k 5 46 61

3 Answers

- In practice, these tend to be implemented with CAS and LL/SC. (...and some spinning before giving up the time slice of the thread - usually by calling into a kernel function that switches context.)
- If you only need a spinlock, wikipedia gives you an example which trades CAS for lock prefixed `xchg` on x86/x64. So in a strict sense, a CAS is not needed for crafting a spinlock - but some kind of atomicity is still required. In this case, it makes use of an atomic operation that can write a register to memory and return the previous contents of that memory slot in a *single step*. (To clarify a bit more: the *lock* prefix asserts the #LOCK signal that ensures that the current CPU has exclusive access to the memory. On todays CPUs it is not necessarily carried out this way, but the effect is the same. By using `xchg` we make sure that we will not get preempted somewhere between reading and writing, since instructions will not be interrupted half-way. So if we had an imaginary *lock mov reg0, mem* / *lock mov mem, reg1* pair (which we don't), that would not quite be the same - it could be preempted just between the two movs.)
- On current architectures, as pointed out in the comments, you mostly end up using the atomic primitives of the CPU and the coherency protocols provided by the memory subsystem.
- For this reason, you not only have to use these primitives, but also account for the cache/memory coherency guaranteed by the architecture.
- There may be implementation nuances as well. Considering e.g. a spinlock:
  - instead of a naive implementation, you should probably use e.g. a TTAS spin-lock with some exponential backoff,
  - on a Hyper-Threaded CPU, you should probably issue `pause` instructions that serve as hints that you're spinning - so that the core you are running on can do something

useful during this

- you should really give up on spinning and yield control to other threads after a while
- etc...

- this is still user mode - if you are writing a kernel, you might have some other tools that you can use as well (since you are the one that schedules threads and handles/enables/disables interrupts).

edited Mar 7 '10 at 14:27

answered Mar 3 '10 at 1:17

Andras Vass
8,276 18 37

2 To extend this, CAS and similar operations are used to implement synchronization because CPUs are specifically designed to have them be *atomic* operations - they do everything in a single step, without any other operation being able to interrupt them. – Amber Mar 3 '10 at 1:36

Note: As **John Knoeller** has pointed out, `xchg` implies a *lock* starting with the 80386 - the prefix is written in most samples for clarity (which I think is a good practice), not out of necessity. This is not true for the others - e.g. `cmpxchg` . Therefore I think it is the safest to always explicitly specify the prefix when you intend to gain exclusive access to the memory. – Andras Vass Mar 3 '10 at 7:15

The x86 architecture, has long had an instruction called `xchg` which will exchange the contents of a register with a memory location. xchg has always been atomic.

There has also always been a `lock` prefix that could be applied to ~~any~~ a single instruction to make that instruction atomic. Before there were multi processor systems, all this really did was to prevent an interrupt from being delivered in the middle of a locked instruction. (xchg was implicitly locked).

This article has some sample code using xchg to implement a spinlock
http://en.wikipedia.org/wiki/Spinlock

When multi CPU and later multi Core systems began to be built, more sophisticated systems were needed to insure that lock and xchg would synchronize all of the memory subsystems, including l1 cache on all of the processors. About this time, new research into locking and lockless algorithms showed that atomic CompareAndSet was a more flexible primitive to have, so more modern CPUs have that as an instruction.

Addendum: In comments **andras** supplied a "dusty old" list of instructions which allow the `lock` prefix. http://pdos.csail.mit.edu/6.828/2007/readings/i386/LOCK.htm

edited Mar 3 '10 at 3:48

answered Mar 3 '10 at 2:02

John Knoeller
21.5k 2 18 63

@andras: Yes, I guess that was misleading, I'll change the wording. And thank you for the list. – John Knoeller Mar 3 '10 at 3:41

I like to think of thread synchronization as a bottom up where processor and operating system provide construct that are primitive to more sophisticated

At the processor level you have CAS and LL/SC which allow you to perform a test and store in a single atomic operation ... you also have other processor constructs that allow you to disable and enable interrupt (however they are considered dangerous ... under certain circumstances you have no other option but to use them)

operating system provides the ability to context switch between tasks which can happen every time a thread has used its time slice ... or it can happen due to otgher reasons (I will come to that)

then there are higher level constructs like mutexes which uses these primitive mechanisms provided by processor (think spinning mutex) ... which will continuously wait for the condition to become true and checks for that condition atomically

then these spinning mutex can use the functionality provided by OS (context switch and system calls like yield which relinquishes the control to another thread) and gives us mutexes

these constructs are further utilized by higher level constructs like conditional variables (which can keep track of how many threads are waiting for the mutex and which thread to allow first when the mutex becomes available)

These constructs than can be further used to provide more sophisticated synchronization constructs ... example : semaphores etc

answered Mar 3 '10 at 2:32

jsshah
1,174 7 15