How to guarantee 64-bit writes are atomic?

When can 64-bit writes be guaranteed to be atomic, when programming in C on an Intel x86-based platform (in particular, an Intel-based Mac running MacOSX 10.4 using the Intel compiler)? For example:

```c
unsigned long long int y;
y = 0xfedcba87654321ULL;
/* ... a bunch of other time-consuming stuff happens... */
y = 0x12345678abcdefULL;
```

If another thread is examining the value of y after the first assignment to y has finished executing, I would like to ensure that it sees either the value 0xfedcba87654321 or the value 0x12345678abcdef, and not some blend of them. I would like to do this without any locking, and if possible without any extra code. My hope is that, when using a 64-bit compiler (the 64-bit Intel compiler), on an operating system capable of supporting 64-bit code (MacOSX 10.4), that these 64-bit writes will be atomic. Is this always true?

c multithreading osx atomic lock-free

edited Oct 30 '09 at 2:42

community wiki
3 revs, 3 users 100%
vok

7 Answers

Your best bet is to avoid trying to build your own system out of primitives, and instead use locking unless it **really** shows up as a hot spot when profiling. (If you think you can be clever and avoid locks, don't. You aren't. That's the general "you" which includes me and everybody else.) You should at minimum use a spin lock, see spinlock(3). And whatever you do, **don't** try to implement "your own" locks. You will get it wrong.

Ultimately, you need to use whatever locking or atomic operations your operating system provides. Getting these sorts of things **exactly right** in **all cases** is **extremely difficult**. Often it can involve knowledge of things like the errata for specific versions of specific processor. ("Oh, version 2.0 of that processor didn't do the cache-coherency snooping at the right time, it's fixed in version 2.0.1 but on 2.0 you need to insert a `NOP`.") Just slapping a `volatile` keyword on a variable in C is almost always insufficient.

On Mac OS X, that means you need to use the functions listed in atomic(3) to perform truly atomic-across-all-CPUs operations on 32-bit, 64-bit, and pointer-sized quantities. (Use the latter for any atomic operations on pointers so you're 32/64-bit compatible automatically.) That goes whether you want to do things like atomic compare-and-swap, increment/decrement, spin locking, or stack/queue management. Fortunately the spinlock(3), atomic(3), and barrier(3) functions should all work correctly on all CPUs that are supported by Mac OS X.

answered Sep 16 '08 at 23:35
Chris Hanson
36.7k 5 60 93

5 Thank you for such a warm and fuzzy place to send future 'practical' lock free evangelicals :) + 10 if I could. – Tim Post ♦ Oct 30 '09 at 2:56

On x86_64, both the Intel compiler and gcc support some intrinsic atomic-operation functions. Here's gcc's documentation of them: http://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html

The Intel compiler docs also talk about them here: http://softwarecommunity.intel.com/isn/downloads/softwareproducts/pdfs/347603.pdf (page 164 or thereabouts).

answered Sep 16 '08 at 23:26

Allen
3,618 10 26

On Intel MacOSX, you can use the built-in system atomic operations. There isn't a provided atomic get or set for either 32 or 64 bit integers, but you can build that out of the provided CompareAndSwap. You may wish to search XCode documentation for the various OSAtomic functions. I've written the 64-bit version below. The 32-bit version can be done with similarly named functions.

```
#include <libkern/OSAtomic.h>
// bool OSAtomicCompareAndSwap64Barrier(int64_t oldValue, int64_t newValue, int64_t
*theValue);

void AtomicSet(uint64_t *target, uint64_t new_value)
{
        while (true)
        {
                uint64_t old_value = *target;
                if (OSAtomicCompareAndSwap64Barrier(old_value, new_value, target)) return;
        }
}

uint64_t AtomicGet(uint64_t *target)
{
        while (true)
        {
                int64 value = *target;
                if (OSAtomicCompareAndSwap64Barrier(value, value, target)) return value;
        }
}
```

Note that Apple's OSAtomicCompareAndSwap functions atomically perform the operation:

```
if (*theValue != oldValue) return false;
*theValue = newValue;
return true;
```

We use this in the example above to create a Set method by first grabbing the old value, then attempting to swap the target memory's value. If the swap succeeds, that indicates that the memory's value is still the old value at the time of the swap, and it is given the new value during the swap (which itself is atomic), so we are done. If it doesn't succeed, then some other thread has interfered by modifying the value in-between when we grabbed it and when we tried to reset it. If that happens, we can simply loop and try again with only minimal penalty.

The idea behind the Get method is that we can first grab the value (which may or may not be the actual value, if another thread is interfering). We can then try to swap the value with itself, simply to check that the initial grab was equal to the atomic value.

I haven't checked this against my compiler, so please excuse any typos.

You mentioned OSX specifically, but in case you need to work on other platforms, Windows has a number of Interlocked* functions, and you can search the MSDN documentation for them. Some of them work on Windows 2000 Pro and later, and some (particularly some of the 64-bit functions) are new with Vista. On other platforms, GCC versions 4.1 and later have a variety of __sync* functions, such as __sync_fetch_and_add(). For other systems, you may need to use assembly, and you can find some implementations in the SVN browser for the HaikuOS project, inside src/system/libroot/os/arch.

answered Sep 16 '08 at 23:49

Bob Rost

According to Chapter 7 of Part 3A - System Programming Guide of Intel's processor manuals, quadword accesses will be carried out atomically if aligned on a 64-bit boundary, on a Pentium or newer, and unaligned (if still within a cache line) on a P6 or newer. You should use `volatile` to ensure that the compiler doesn't try to cache the write in a variable, and you may need to use a memory fence routine to ensure that the write happens in the proper order.

If you need to base the value written on an existing value, you should use your operating system's Interlocked features (e.g. Windows has InterlockedIncrement64).

answered Sep 16 '08 at 23:39
Mike Dimmick
6,481 1 8 29

To be even more specific, it is stated in §8.8.1 on page 325. – user405725 Oct 27 '13 at 6:43

On X86, the fastest way to atomically write an aligned 64-bit value is to use FISTP. For unaligned values, you need to use a CAS2 (_InterlockedExchange64). The CAS2 operation is quite slow due to BUSLOCK though so it can often be faster to check alignment and do the FISTP version for aligned addresses. Indeed, this is how the Intel Threaded building Blocks implements Atomic 64-bit writes.

answered Oct 30 '09 at 19:33 community wiki
Adisak

If you want to do something like this for interthread or interprocess communication, then you need to have more than just an atomic read/write guarantee. In your example, it appears that you want the values written to indicate that some work is in progress and/or has been completed. You will need to do several things, not all of which are portable, to ensure that the compiler has done things in the order you want them done (the volatile keyword may help to a certain extent) and that memory is consistent. Modern processors and caches can perform work out of order unbeknownst to the compiler, so you really need some platform support (ie., locks or platform-specific interlocked APIs) to do what it appears you want to do.

"Memory fence" or "memory barrier" are terms you may want to research.

answered Sep 16 '08 at 23:41
Michael Burr
194k 24 264 492

GCC has intrinsics for atomic operations; I suspect you can do similar with other compilers, too. Never rely on the compiler for atomic operations; optimization will almost certainly run the risk of making even obviously atomic operations into non-atomic ones unless you explicitly tell the compiler not to do so.

answered Sep 16 '08 at 23:18
Dark Shikari
6,373 1 18 33