How does InterlockedIncrement work internally?

devblogs.microsoft.com/oldnewthing/20130913-00

September 13, 2013

Raymond Chen

The Interlocked family of functions perform atomic operations on memory. How do they do it?

It depends on the underlying CPU architecture. For some CPUs, it's easy: The x86, for example, has direct support for many interlocked operations by means of the `LOCK` prefix (with the bonus feature that `LOCK` is implied for the `XCHG` opcode.) The ia64 and x64 also have direct support for atomic load-modify-store operations.

Most other architectures break the operation into two parts, known as Load-link/store-conditional. The first part (load-link) reads a value from memory and instructions the processor to monitor the memory address to see if any other processors modify that same memory. The second part (store-conditional) stores a value to memory provided that no other processors have written to the memory in the meantime. An atomic load-modify-store operation is therefore performed by reading the value via load-link, performing the desired computation, then attempting a store-conditional. If the store-conditional fails, then start all over again.

```
LONG InterlockedIncrement(LONG volatile *value)
{
  LONG lOriginal, lNewValue;
  do {
    // Read the current value via load-link so we will know if
    // somebody has modified it while we weren't looking.
    lOriginal = load_link(value);
    // Calculate the new value
    lNewValue = lOriginal + 1;
    // Store the value conditionally. This will fail if somebody
    // has updated the value in the meantime.
  } while (!store_conditional(value, lNewValue));
  return lNewValue;
}
```

(If this looks familiar, it should. You've seen this pattern before.)

Now, asking the CPU to monitor a memory address comes with its own gotchas. For one thing, the CPU can monitor only one memory address at a time, and its memory is very short-term. If your code gets pre-empted or if a hardware interrupt comes in after your `load_link`, then your `store_conditional` will fail because the CPU got distracted by the shiny object known as *hardware interrupt* and totally forgot about that memory address it was supposed to be monitoring. (Even if it managed to remember it, it won't remember it for long, because the hardware interrupt will almost certainly execute its own `load_link` instruction, thereby replacing the monitored address with its own.)

Furthermore, the CPU might be a little sloppy in its monitoring and monitor not the address itself but the cache line. If somebody modifies a different memory location which happens to reside in the same cache line, the `store_conditional` might fail even though you would expect it to succeed. The ARM architecture allows a processor to be so sloppy that any write in the same block of 2048 bytes can cause a `store_conditional` to fail.

What this means for you, the assembly-language coder who is implementing an interlocked operation, is that you need to minimize the number of instructions between the `load_link` and `store_conditional`. For example, `InterlockedIncrement` merely adds 1 to the value. The more instructions you insert between the `load_link` and `store_conditional`, the greater the chance that your `store_conditional` will fail and you will have to retry. And if you put too much code in between, your `store_conditional` will *never* succeed. As an extreme example, if you put code that takes five seconds to calculate the new value, you will certainly receive several hardware interrupts during those five seconds, and your `store_conditional` will always fail.

Bonus reading: Why did InterlockedIncrement/Decrement only return the sign of the result?

Raymond Chen

Follow