FRIDAY, MAY 18, 2012

Peeking Under the Hood

Back in the day, if you asked a computer programmer (because that is what we were called way back then) what language they programmed in you'd get answers like FORTRAN, COBOL, and for the proud few maybe assembler. Today, when you ask that question, it's anybody's guess what the answer might be, but there's a good chance it would be a domain specific language I've never heard of.

Both then and now, if you asked a random software developer how did their language perform execution synchronization among multiple threads of control (or tasks, or processes, or execution contexts, or whatever damn thing their problem domain wants to call it), the typical answer would be "I dunno" or "isn't that the operating system's job?" or "what's a task?".

But most developers at least understand the basic problem in the abstract. The classic pedagogical example is your checking account. If two people are each independently trying to cash checks for $100 that you wrote when your account contains just $150, somehow the action of *reading* your balance to check for sufficient funds and then *modifying* your balance to withdraw the money has to be indivisible. Otherwise person A could check for sufficient funds, then person B could check, then person A gets their money, then person B, even though your account doesn't actually have of enough cash to cover both checks.

This wasn't a problem when there was only one bank, in one building, with one ledger book. The ledger was the final arbiter of how much money you had. But as soon as your bank had more than one teller, then more than one building, then maybe not even in the same city, suddenly the *instantaneous* balance of your checking account was a little more ambiguous than either your bank or your creditors were comfortable with. Even with modern communications, speed of light being what it is.

This classic pedagogical example is actually pretty crappy, because in real-life it happens all the time. Either the bank overdraws your account and then sends some leg breakers to your house, or in the case of electronic transfers, the transfer is held pending until the bank's central computer can figure out whether or not you're a deadbeat writing cold checks. In the former case the bank tries to fix the problem after the fact. In the latter case they're deferring the problem until they've had some time to ponder it. (This is also why you can order something online from Amazon.com but get email later saying "Oops, sorry, it's actually out of stock.") But in principle anyway, the example points out the issue in a context that is pretty easily understood.

So when developers understand this problem in the abstract (and most do), and if they've done some development in a language like Java that natively supports multithreading, their answer regarding execution synchronization might be "use the `synchronize` keyword". If they have used a POSIX thread library, it might be "use a mutex" (where "mutex" is used by those in the know as shorthand for *mutual exclusion*).

METADATA

CHIP OVERCLOCK

DENVER, COLORADO, UNITED STATES

Chip Overclock® is a megalomaniacal supervillain and the secret identity, pseudonym, alter ego, and registered trademark of John Sloan, a technology product developer who specializes in very large and very small systems on the internet of interesting things: distributed, real-time, high performance, high altitude, embedded, concurrent, parallel, asynchronous, remote, close to bare metal.

VIEW MY COMPLETE PROFILE

COPYRIGHT

TRADEMARKS

LABELS

AI (8)

Android (13)

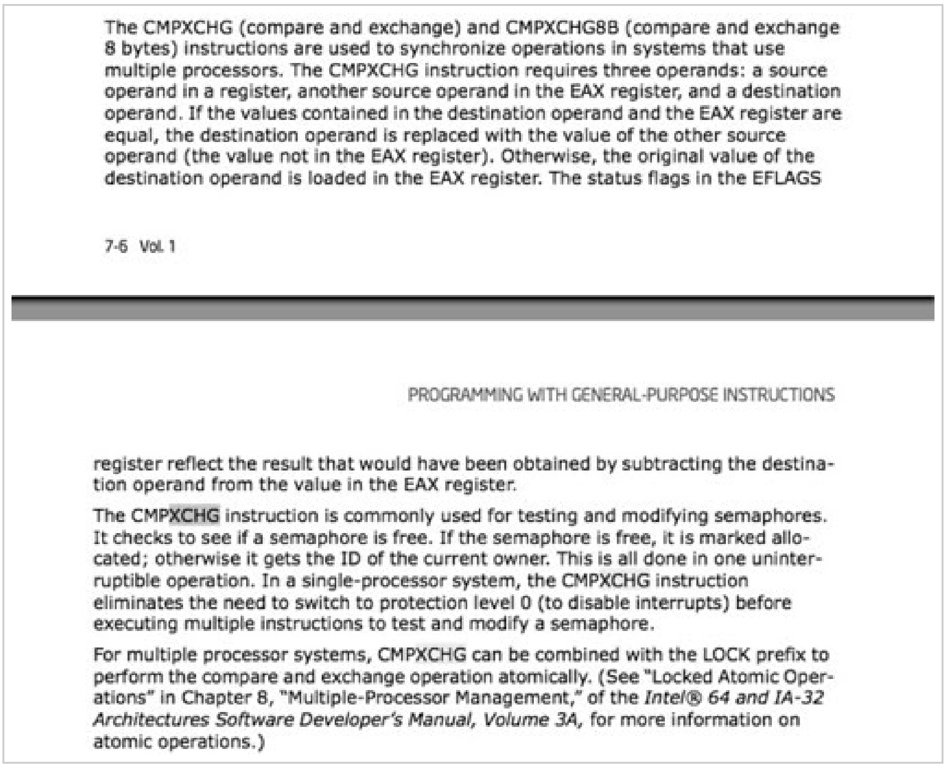
If they know what SQL stands for, they might say "use a lock" or "a transaction". If they are really really old (but not as old as me) they might say "use a semaphore" or even "a monitor".

These are all software mechanisms that prevent more than one thread of control from executing a function or reading and modifying a variable simultaneously, or (as those in the know like to say) *concurrently*. In our canonical pedagogical example, each teller checking your bank balance was a separate thread of control operating concurrently.

But those answers just beg the question. When asked "how does the `synchronize` keyword (or semaphore, etc.) work?" typically we're back to "I dunno" or "it's the operating system's problem". If you keep asking "How?", peeling back layers and layers of software, down through the application, through the programming language, the library, the operating system, eventually you will discover that there is some mechanism down near the level of bare metal, often a single machine instruction, that actually does the heavy lifting of execution synchronization, insuring that two threads of control can't read and modify the same location location in memory by each reading it and then each modifying it, just like the checking account. All those other software synchronization mechanisms are ultimately built on top of this one low level hardware synchronization mechanism.

It was not always thus. I remember stories in the 1970s, when I first started having to worry about this stuff, about how the IBM mainframe development folks who were looking at the very earliest multiprocessor designs figured out they needed what generically became known as a *test and set* instruction. *Test and set* was a machine instruction that simultaneously remembered what the value of a memory location was and changed its value, and it did those two operations indivisibly, or (by those in the know) *atomically*. It was the machine instruction equivalent to the checking account example, checking the balance and deducting if funds were sufficient, except it was operating on just a byte or word of memory. This machine instruction was used to insure that, many layers of software above, code that was in fact checking your bank balance and deducting funds did so safely by allowing only one thread of control to do so at a time.

Eventually on the IBM mainframe the *test and set* instruction was replaced with the *compare and swap* instruction because it solved a larger class of execution synchronization problems. But to this day, decades later, machine instructions or their equivalents, on widely different processor architectures, used for implementing mutual exclusion operators like semaphores, are referred to as *test and set*, *compare and swap*, or something similar.



The CMPXCHG (compare and exchange) and CMPXCHG8B (compare and exchange 8 bytes) instructions are used to synchronize operations in systems that use multiple processors. The CMPXCHG instruction requires three operands: a source operand in a register, another source operand in the EAX register, and a destination operand. If the values contained in the destination operand and the EAX register are equal, the destination operand is replaced with the value of the other source operand (the value not in the EAX register). Otherwise, the original value of the destination operand is loaded in the EAX register. The status flags in the EFLAGS

7-6 Vol. 1

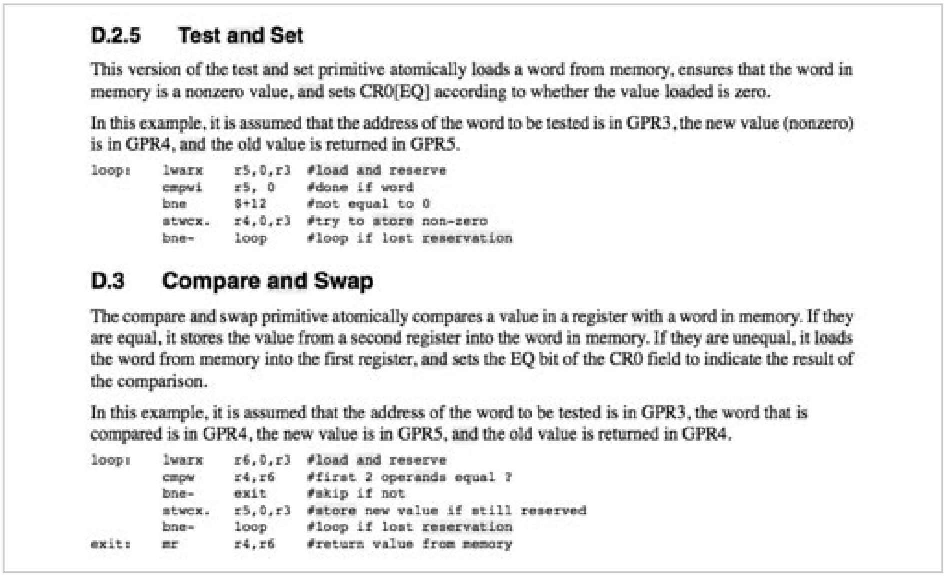
PROGRAMMING WITH GENERAL-PURPOSE INSTRUCTIONS

register reflect the result that would have been obtained by subtracting the destination operand from the value in the EAX register.

The CMPXCHG instruction is commonly used for testing and modifying semaphores. It checks to see if a semaphore is free. If the semaphore is free, it is marked allocated; otherwise it gets the ID of the current owner. This is all done in one uninterruptible operation. In a single-processor system, the CMPXCHG instruction eliminates the need to switch to protection level 0 (to disable interrupts) before executing multiple instructions to test and modify a semaphore.

For multiple processor systems, CMPXCHG can be combined with the LOCK prefix to perform the compare and exchange operation atomically. (See "Locked Atomic Operations" in Chapter 8, "Multiple-Processor Management," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A,* for more information on atomic operations.)

(Above: Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual", Volume 1, 325462-043US, May 2012, pages 7-6 through 7-7)



D.2.5 Test and Set

This version of the test and set primitive atomically loads a word from memory, ensures that the word in memory is a nonzero value, and sets CR0[EQ] according to whether the value loaded is zero.

In this example, it is assumed that the address of the word to be tested is in GPR3, the new value (nonzero) is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3 #load and reserve
        cmpwi   r5, 0   #done if word
        bne     $+12    #not equal to 0
        stwcx.  r4,0,r3 #try to store non-zero
        bne-    loop    #loop if lost reservation
```

D.3 Compare and Swap

The compare and swap primitive atomically compares a value in a register with a word in memory. If they are equal, it stores the value from a second register into the word in memory. If they are unequal, it loads the word from memory into the first register, and sets the EQ bit of the CR0 field to indicate the result of the comparison.

In this example, it is assumed that the address of the word to be tested is in GPR3, the word that is compared is in GPR4, the new value is in GPR5, and the old value is returned in GPR4.

```
loop:   lwarx   r6,0,r3 #load and reserve
        cmpw    r4,r6   #first 2 operands equal ?
        bne-    exit    #skip if not
        stwcx.  r5,0,r3 #store new value if still reserved
        bne-    loop    #loop if lost reservation
exit:   mr      r4,r6   #return value from memory
```

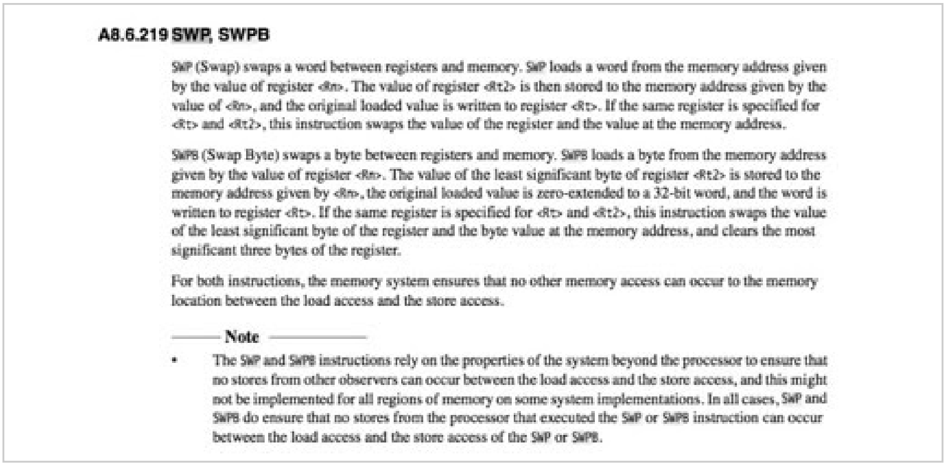
(Above: Freescale Semiconductor, "Programming Environments Manual for 32-bit Implementations of the PowerPC Architecture", Revision 3, MPCFPE32B, September 2005, page D-3)

ARCHIVE

▶ 2024 (9)
▶ 2023 (18)
▶ 2022 (15)
▶ 2021 (17)
▶ 2020 (27)
▶ 2019 (8)
▶ 2018 (20)
▶ 2017 (19)
▶ 2016 (6)
▶ 2015 (7)
▶ 2014 (20)
▶ 2013 (13)
▼ 2012 (47)
  ▶ December (5)
  ▶ September (1)
  ▶ August (1)
  ▶ July (4)
  ▶ June (6)
  ▼ May (5)
    Updates
    The C++ Pointer to Member Operators
    Peeking Under the Hood
    Brain Versus Heart
    The Code Also Rises
  ▶ April (3)
  ▶ March (3)
  ▶ February (8)
  ▶ January (11)
▶ 2011 (40)
▶ 2010 (12)
▶ 2009 (8)
▶ 2008 (25)
▶ 2007 (56)
▶ 2006 (16)

BLOGS

Alastair Cockburn
All Things Distributed
Better Embedded System SW
Coding Horror
Coding Relic
E/S and I



A8.6.219 SWP, SWPB

SWP (Swap) swaps a word between registers and memory. SWP loads a word from the memory address given by the value of register <Rn>. The value of register <Rt2> is then stored to the memory address given by the value of <Rn>, and the original loaded value is written to register <Rt>. If the same register is specified for <Rt> and <Rt2>, this instruction swaps the value of the register and the value at the memory address.

SWPB (Swap Byte) swaps a byte between registers and memory. SWPB loads a byte from the memory address given by the value of register <Rn>. The value of the least significant byte of register <Rt2> is stored to the memory address given by <Rn>, the original loaded value is zero-extended to a 32-bit word, and the word is written to register <Rt>. If the same register is specified for <Rt> and <Rt2>, this instruction swaps the value of the least significant byte of the register and the byte value at the memory address, and clears the most significant three bytes of the register.

For both instructions, the memory system ensures that no other memory access can occur to the memory location between the load access and the store access.

Note

- The SWP and SWPB instructions rely on the properties of the system beyond the processor to ensure that no stores from other observers can occur between the load access and the store access, and this might not be implemented for all regions of memory on some system implementations. In all cases, SWP and SWPB do ensure that no stores from the processor that executed the SWP or SWPB instruction can occur between the load access and the store access of the SWP or SWPB.

(Above: ARM, "ARM Architecture Reference Manual", ARM v7-A and ARM V7-R Edition, ID120611, 2011, page A8-432)

Recent versions of the GNU C compiler even have a built-in function called

```
__sync_lock_test_and_set()
```

that attempts to generate code for the underlying processor using whatever that hardware's native mechanism is at the machine code level, in a kind of homage to this decades old invention, now in a portable open source form.

Sometimes there is an otherwise innocuous machine instruction that just happens to be perfectly useful as *test and set*. For example, the DEC PDP-11 minicomputer had a *decrement* or `dec` instruction which atomically altered memory and set the condition code bits in the global hardware status register. You could set a memory location representing a lock to a value larger than zero to represent "unlocked", the value zero to represent "just locked", and a negative number to mean "already locked by someone else". You would use the `dec` instruction to decrement a word in memory and afterwards check the condition code by using a branch on condition instruction; if the result was zero, that meant the value had been positive before your decrement, hence you now own the resource that it represents; if the result was negative, someone had already locked it. You unlocked the memory location by using the *increment* or `inc` instruction. The value of the word in memory could be interpreted as a count of the number of resources available, so values larger than one could be used to implement a *counting semaphore* instead of just a *mutex semaphore*, representing, say, a pool of buffers. I have no idea if the PDP-11 hardware architects intended for `inc` and `dec` to be used this way, but it seems likely.

Sometimes there is no hardware execution synchronization mechanism at all. The IBM folks really only needed the *test and set* instruction when they started looking at multiprocessor systems. Before then, the only concurrency that could occur within a uniprocessor system was with an interrupt service routine or ISR. A hardware signal interrupts the normal flow of instruction execution and control is automatically passed to a special function or ISR that handles whatever the signal represents, like "a character is available because someone typed it on the keyboard". It is essentially a case of the hardware calling a subroutine. It is how input and output (and lots of other *asynchronous* and *real-time* stuff) is handled in most modern hardware architectures. This could result in the same kinds of concurrency issues if both the ISR and the code it interrupted read and modified the same variable in memory. In the uniprocessor cases, it was merely necessary to temporarily disable interrupts when the software knew it was

Embedded Apprentice
Embedded Artistry
Embedded Gurus
Embedded in Academia
Escape from Cubicle Nation
Fazal Majid's low-intensity blog
Flip Chart Fairy Tales
Gripe Line
High Scalability
How to Change the World
Huan Liu
Jack Ganssle
Joel on Software
Keep the Joint Running
Les Hatton
Pete Warden
Process Perfection
rachelbythebay
ribbonfarm
Robert Cringely
Technology ViewPoint
The Cranky Project Manager
Work Matters

LINKS

Chip Overclock
Digital Aggregates Corporation
GitHub

SLIDES

Memory Models
Internet Protocol version 6
Post Moore's Law

VIDEOS

Balance Wheel Watch
Internet Protocol version 6
Pendulum Clock
Pendulum Escapement
Time and Space

TOOLS

Coast Guard GPS Status Reports
FAA Defense Internet NOTAMs

ANALYTICS

in section of code in which it was critical that a sequence of operations, like reading and modifying a particular variable, be done atomically. Such a segment of code is called (those in the know again) a *critical section*.

Google

This technique persists today in, for example on the tiny Atmel megaAVR microcontrollers I've been noodling with recently. Since they are uniprocessors, it is enough to disable interrupts going into a critical section using the `cli` machine instruction that clears the system-wide interrupt enable bit (the `I-bit`) in the system status register (`SREG`), and to reenable interrupts when exiting the critical section by using the `sei` machine instruction that sets the system-wide interrupt enable bit.

But that's not typically what megaAVR code does. If you have nested critical sections (easily done when you're calling subroutines from inside a critical section and they too have their own critical sections), you don't want to reenable interrupts when you exit the inner critical section because they weren't enabled when you first entered the inner critical section. You want to return the interrupt state to whatever it was before you entered your critical section, understanding that it might have already been disabled by whoever called you. So most megaAVR code (including mine) implements critical sections in C like this:

```
unsigned char temp = SREG;
cli();


/* Critical section code goes here. */


SREG = temp;
```

Entering the critical section, we read and store `SREG` (which is memory mapped, meaning hardware allows us to access it like a memory location; other processor architectures have a special machine instruction just to do this) containing the `I-bit` in a temporary variable, and then we disable interrupts by clearing the `I-bit` in `SREG` using `cli()`. Exiting the critical section, we don't enable interrupts by setting the `I-bit` using `sei()`. We instead restore `SREG` using the saved copy in our temporary variable, returning the `I-bit` to whatever its prior value was. It will have been a one (1) if the caller was not already in a critical section, zero (0) if it was.

So, are we good?

No, not really. You may have noticed that there's actually an issue here. If an ISR were to interrupt this code snippet *after* the `SREG` has been read, but *before* `cli()` disables interrupts -- which is entirely possible, because by definition we haven't disabled interrupts yet -- the ISR could change the value of `SREG` (there are after all seven other bits in SREG that mean other things). When we restore the value of `SREG` as we exit our critical section, we're restoring it to its value before we read it, not the value to which the ISR modified it after we read it. This is known as a *race condition*: two separate threads of control coming into conflict over the value of a shared variable because their use of it was not properly synchronized. Ironically, though, this race condition occurs in our synchronization mechanism, the very mechanism intended to prevent race conditions from occurring. (It's called a race condition because the outcome is timing dependent and hence *non-deterministic*: it depends on which thread of control wins the *data race* to change the variable.)

How does the megaAVR solve this problem? I don't think it does. In practice, this typically isn't an issue. I've never had a reason for an ISR to modify the value of `SREG`. Although I should point out that every ISR *implicitly* modifies `SREG`: the megaAVR hardware automatically clears the `I-flag` when entering an ISR, and the return from interrupt (`reti`) machine instruction automatically sets the `I-flag` when the ISR exits. This is so an ISR cannot itself be interrupted. Unlike other architectures (like my beloved PDP-11) this is not done by saving and restoring `SREG` on the stack, but by actually altering the `I-bit` in the `SREG`. This is possible because the ISR could not have been entered in the first place had the `I-bit` not already been set. But it still behooves the developer not to otherwise alter `SREG` inside the ISR lest wackiness ensue.

When you've spent enough time writing code close to bare metal like I have, you too will begin to look with skepticism at even the lowest level of operations of whatever processor you are using, asking yourself "how does this really work under the hood?" You will find yourself spending hours peering at eight hundred page processor reference manuals. I call that a good thing. Your mileage may vary.

POSTED BY CHIP OVERCLOCK AT 1:52:00 PM

LABELS: ARDUINO, C, C++, CONCURRENCY, EMBEDDED SYSTEMS

NO COMMENTS:

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

CLOUD

EMBEDDED SYSTEMS BUSINESS LINUX/GNU SOFTWARE DEVELOPMENT C MISCELLANEOUS DISTRIBUTED SYSTEMS GEOLOCATION PRODUCT DEVELOPMENT RASPBERRY PI C++ HOROLOGY SPECULATION ARDUINO CONCURRENCY FAILURE ANDROID HISTORY MEASUREMENT DYSFUNCTION NUTS AND BOLTS MEMORY MODELS STORAGE SUPERCOMPUTING AI BEAGLEBOARD JAVA TRAFFIC MANAGEMENT AR.DRONE S3 SAFETY SECURITY TELEPHONY IOT NATIONAL SECURITY REVERSE ENGINEERING ASTERISK CAREER ODROID POSIX WEB SYSTEMS ENTROPY PHYSICS TALKS CARTOGRAPHY MAC POLITICS RIGHT TO REPAIR SATIRE CULTURE IPHONE

SEARCH THIS BLOG

Search