IBM

Save your code from meltdown using PowerPC atomic instructions

Write correct, concurrent code

Level: Introductory

Jonathan Rentzsch (jon.dw@redshed.net), President, Red Shed Software

02 Nov 2004

Something as simple as incrementing an integer can fail in a concurrent environment. This article illustrates the failure scenario and introduces the PowerPC's coping mechanism: atomic instructions. Learn how to use these assembly-level instructions to update memory correctly, even in the face of concurrency.

Pop quiz! What's wrong with the C code in Listing 1, which increments an integer?

Listing 1. Incrementing an integer: a subtle flaw

```
++gConnectCount;
```

Time's up! Did you spot the flaw? If not, don't feel too badly -- it was a bit of a trick question. Just by itself, this simple code isn't wrong. However, you can place it into a context where this code will produce incorrect results: randomly losing increments. That is, calling `++gConnectCount` *may not actually increment* `gConnectCount` -- and that's troubling.

The trouble starts if `++gConnectCount` is *reentered*. This can happen if the code is run while it's already running, for instance, if it's called from a signal handler, or from another thread. Even on a single-processor machine, a preemptive threading architecture can reenter code while a thread is already running.

It seems odd to talk about one line of code being *reentered*. C is a rather low-level language, with most of its semantics mapping onto processor primitives one-to-one. So, thinking that one line of code incrementing a counter would result in one processor instruction generated is not outrageous. Indeed, this is the case for CISC architectures like x86 and 68K.

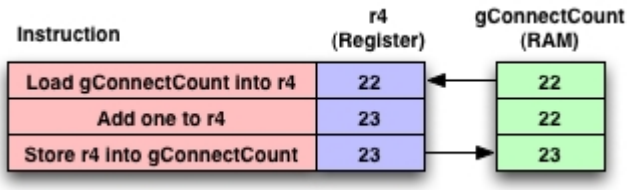
The IBM® PowerPC®, however, is a RISC architecture with its characteristic load-store design -- so the PowerPC doesn't have instructions that perform operations directly on memory. Instead, you *load* the memory into a register, perform the operation, and then *store* it back. Thus, your precious C compiler translated `++gConnectCount` into something like four PowerPC instructions, as Listing 2 shows.

Listing 2. Incrementing an integer: generated assembly

```
lwz  r3, gConnectCount(rtoc) // Load address of gConnectCount into register r3.
lwz  r4, 0(r3)               // Load integer from RAM into register r4.
addi r4, r4, 1               // Add 1 to register r4.
stw  r4, 0(r3)               // Store incremented value from register r4 back into RAM.
```

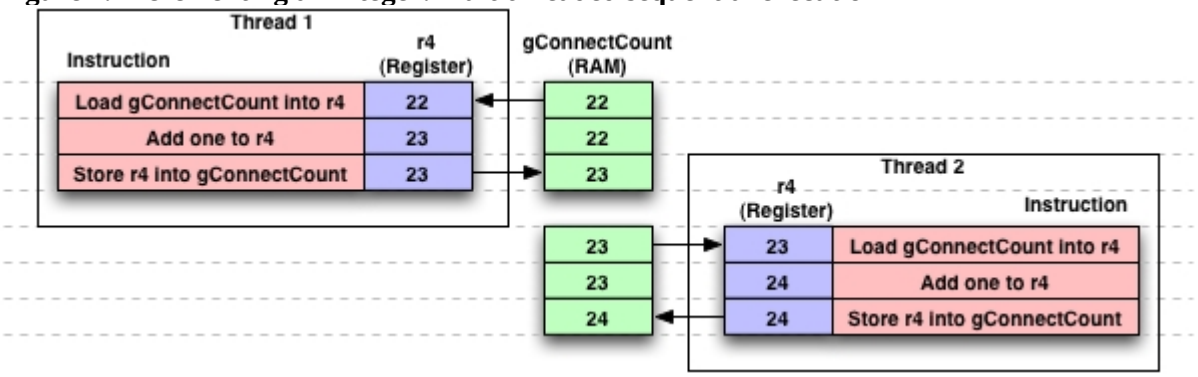
You can pretty much ignore Listing 2's first line -- it just fetches the address of the `gConnectCount` global variable into register r3. The real meat is in the `lwx`/`addi`/`stw` triplet. Figure 1 shows this code as an illustration, presenting what happens when `gConnectCount`'s value starts out at `22`.

Figure 1. Incrementing an integer using load/store



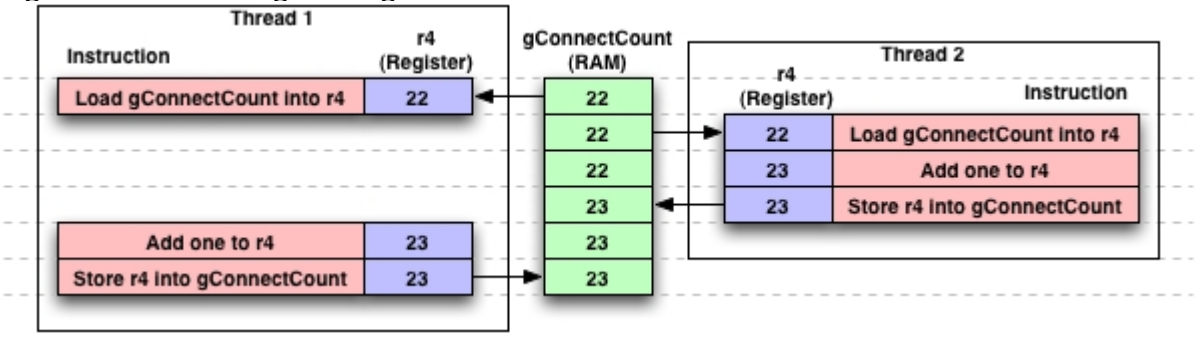
As you can see, it takes three separate instructions to increment an integer in RAM. Since it takes more than one instruction to accomplish the task, this sequence will possibly be interrupted, as is the case in time-slice-based preemptive multithreading. However, multithreading by itself isn't the problem. For example, it's okay if the code is executed sequentially by multiple threads, as Figure 2 illustrates.

Figure 2. Incrementing an integer: multithreaded sequential execution



The fun starts when this sequence is interrupted and reentered, as Figure 3 illustrates.

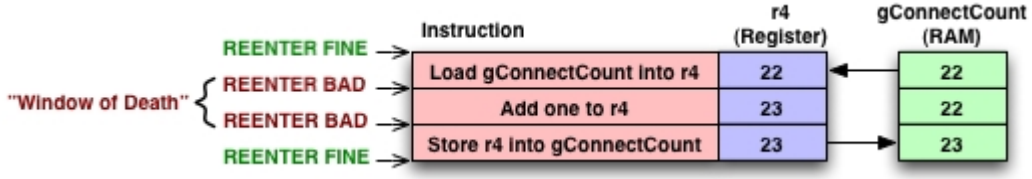
Figure 3. Incrementing an integer: multithreaded concurrent execution



The tragedy unfolds when two threads overlap and read `gConnectCount` twice. The threads, each unaware of the other's concurrent modification, interact in such a way that one increment is "lost".

Looking again at the sequence of instructions necessary to increment an integer, you can pick out four distinct events, as Figure 4 shows. You can reenter the code before it loads the value (the first event), or after it writes out the result (the last event). However, between the first and last event, there's a span of time when the code will malfunction if reentered. This interval is the *Window of Death*.

Figure 4. Incrementing an integer: the Window of Death



Closing the Window of Death with atomic instructions

The problem with `++gConnectCount` is that it boils down to at least three instructions, and bad things happen if this sequence of instructions is reentered. But `++gConnectCount`

What about sig_atomic_t?

You *can* deal with this, up to a point,

is as tight as you can express incrementing an integer in C. Indeed, you can't make this code reentrant-safe in portable C. For that, you need to drop down and use processor family-specific instructions.

The PowerPC User Instruction Set Architecture provides two interrelated instructions that you can use to make updating memory *atomic* -- that is, indivisible. Once an update is made indivisible, reentering it won't cause any problems. That's because other threads see the shared state before the update, or after the update, but never *during* the update (when data might be inconsistent). These two instructions are Load Word and Reserve Index (`lwarx`) and Store Word Conditional Index (`stwcx.`).

`lwarx` works just like the common Load Word and Zero Indexed (`lwzx`), except that it places a *reservation* on the loaded address in addition to loading the data. The PowerPC processor can hold only one reservation at a time.

Likewise, `stwcx.` works just like any other store instruction. The big difference is that `stwcx.` is *conditional* -- the store is only performed if a reservation is present on the given address. If a reservation exists, then it clears the reservation, performs the store and sets the condition register `CR0[EQ]` to true. Otherwise, the instruction does nothing except set `CR0[EQ]` to false.

in standard C. The C standard provides a type called `sig_atomic_t`, and accesses to this type are atomic.

This doesn't quite get you what you need. An increment is two accesses (a read and a write), so even if the accesses are atomic, the increment might not be. Furthermore, it doesn't help you access any other types, and not every compiler for PowerPC targets actually provides this type (as of this writing). This is a gap in standards conformance, but that doesn't make your code compile.

The atomicity package in Resources shows one way to work around this, which is to write very unportable, processor-specific, reentrant-safe, code with an interface that you can then use from more portable code.

Think of a "reservation" as a kind of register. During each store instruction, the processor compares the given address to the reservation. If they are equal, the reservation is cleared. However, reservations also work in a multiprocessor environment. If processor A places a reservation on address X and processor B stores to address X, processor A's reservation is cleared.

When coupled, these two instructions provide a foundation to implement code that's safe against reentrancy. Listing 3 shows how by rewriting `++gConnectCount` to be atomic.

Listing 3. Incrementing an integer: atomically

```
retry:
  lwarx  r4, 0, r3 // Read integer from RAM into r4, placing reservation.
  addi   r4, r4, 1 // Add 1 to r4.
  stwcx. r4, 0, r3 // Attempt to store incremented value back to RAM.
  bne-   retry     // If the store failed (unlikely), retry.
```

Not surprisingly, the code looks largely the same as before. The load/store instructions (`lwz` and `stw`) have been replaced with their reservation-aware brethren (`lwarx` and `stwcx.`) and looping (the `retry:` label and `bne-` instruction) has been added.

The looping has to be added because the store can now fail (if another thread has preempted the reservation). So, after the conditional store, you should test `CR0[EQ]` to deduce whether the store succeeded. If it did: great, continue on. If it didn't: you should loop back, load from RAM again, and retry the operation. Also, notice the minus symbol after the `bne` instruction. That's a hint, passed through the assembler and encoded into the instruction, informing the branch prediction unit that this branch is unlikely to be taken. This is just an optimization, but it makes sense here -- you don't expect to be interrupted very often in your short three-

Warning: this is nonproduction code!

All the code presented in this article is instructional and not intended for production use. In particular, real-world atomic code tends to have `sync` instructions sprinkled throughout. This is usually to compensate for buggy hardware. This is why, in general, you'll want to use the atomic functions provided by your operating system or libraries instead of writing your own.

instruction window.

The example in this article just increments an integer. However, you can do anything that involves updating a single 32-bit value. With this in mind, you can write a general function wrapper for the atomic instructions, enabling atomic updating from higher-level C code. Listing 4 (which takes advantage of GCC 3.3's new CodeWarrior-style assembly-within-a-C-function support) gives an example.

Listing 4: General atomic updating function

```
  asm
  long         // <- Zero on failure, one on success (r3).
AtomicStore(
  long prev,   // -> Previous value (r3).
  long next,   // -> New value (r4).
  void *addr ) // -> Location to update (r5).
{
retry:
  lwarx  r6, 0, r5 // current = *addr;
  cmpw   r6, r3    // if( current != prev )
  bne    fail      // goto fail;
  stwcx. r4, 0, r5 // if( reservation == addr ) *addr = next;
  bne-   retry     // else goto retry;
  li      r3, 1    // Return true.
  blr              // We're outta here.
fail:
  stwcx. r6, 0, r5 // Clear reservation.
  li     r3, 0     // Return false.
  blr              // We're outta here.
}
```

With your new atomic function wrapper, you can rewrite `++gConnectCount` to be fully reentrant, yet keep it in high-level C (see Listing 5).

Listing 5. Incrementing an integer atomically in C

```
long stored;
do {
    stored = AtomicStore(gConnectCount, gConnectCount + 1, &gConnectCount);
} while(!stored);
```

While this article discusses atomic instructions on PowerPC32/64 processors, the POWER™ line offers a great deal of ISA compatibility with the PowerPC. The concepts outlined here are at least applicable to POWER programming, and chances are the code is as well.

Writing atomic code isn't hard, but it often requires a little esoteric knowledge about the processor you're targeting. The PowerPC offers a clean and high-performance model for handling concurrency correctly, so it's a great place to learn the reentrancy ropes.

Resources

- The PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-bit Microprocessors is the definitive guide to programming for PowerPC processors. Appendix E, "Synchronization Programming Examples" provides sample code for numerous atomic operations.
- Introduction to assembly on the PowerPC provides -- well -- an introduction to PowerPC assembly, of course -- but also useful links in its own Resource section (developerWorks, July 2002).
- The ELF Assembler User's Guide for PowerPC describes how to use the ELF (Executable and Linkable Format) assembler for the PowerPC microprocessor. Hence the name.
- The IBM Assembler Language Reference is written to be specific to AIX, but contains much that is

helpful in other environments as well. See what it has to say on stwcx. and lwarx.

- Whether you're fairly new to PowerPC programming or just curious about some of the nuances of the PowerPC architecture, you may want to check ECS's page of PowerPC Technical Tidbits; it discusses the PPC register model; branching and branch prediction, and more. `lwarx` and `stwcx.` are discussed in the Semaphore Support section.
- If you knew that Edsger Dijkstra was a pioneer in the field of concurrent programming, then you might just be the right person to help fill out Wikipedia's Parallel programming page.
- Atomicity is also important in distributed Grid computing.
- Basic use of pthreads: An introduction to POSIX threads isn't PowerPC-specific, but rather gives an overview of multithreaded programming (developerWorks, January 2004).
- Multiple processors, processors, and threads are a good thing -- but too much of them (without precautions) can lead to race conditions. Learn more in the Secure programmer: Prevent race conditions (developerWorks, October 2004).
- The author developed an atomicity package which works on 68K and PowerPC.
- Have experience you'd be willing to share with Power Architecture zone readers? Article submissions on all aspects of Power Architecture technology from authors inside and outside IBM are welcomed. Check out the Power Architecture author FAQ to learn more.
- Have a question or comment on this story, or on Power Architecture technology in general? Post it in the Power Architecture technical forum or send in a letter to the editors.
- Get a subscription to the Power Architecture Community Newsletter when you Join the Power Architecture community.
- All things Power are chronicled in the developerWorks Power Architecture editors' blog, which is just one of many developerWorks blogs.
- Find more articles and resources on Power Architecture technology and all things related in the developerWorks Power Architecture technology content area.
- Download a Power Architecture Pack to demo a SoC in a simulated environment, or just to explore the fully licensed version of Power Architecture technology.

About the author

Jonathan 'Wolf' Rentzsch runs Red Shed Software, a small Illinois software boutique. He also leads PSIG, a suburban Mac programmer group, and co-hosts CAWUG, a downtown Mac & WebObjects programmer group.
