A nice simple example of a memory barrier requirement

Originally appeared at:
`http://sites.google.com/site/peeterjoot/math2010/atomicSimple.pdf`

Peeter Joot — peeter.joot@gmail.com
June 7, 2010 atomicSimple.tex

---

1. Motivation.

We apparently have some code where a pointer is used to flag that specific updates to a structure can be consumed by other threads. This presents a nice opportunity to document one of the simpler, yet realistic, fashions that memory barriers are required for production code. This description should complement the previous barrier related posts An attempt to illustrate differences between memory ordering and atomic access, and Intel memory ordering, fence instructions, and atomic operations.

2. Guts

I havent seen the code in question, but would imagine it could look something like this:

```
1  struct broadcast
2  {
3      int x ;
4      int y ;
5      // ...
6  } ;
7
8  struct sharedMem
9  {
10     volatile broadcast * publicUnprotectedMemoryWithStuff ;
11 } ;
12
13 /* producer */
14 void updateMemoryAndLetEverybodySeeIt( broadcast * pStuff, sharedMem * pShared )
15 {
16     pStuff->x = 3 ;
17     pStuff->y = 7 ;
18
19     pShared->publicUnprotectedMemoryWithStuff = pStuff ;
20 }
21
22 /* consumer */
23 void consumePublicMemory( broadcast * pStuff )
24 {
25     if ( pStuff->publicUnprotectedMemoryWithStuff )
26     {
27         int v = pStuff->publicUnprotectedMemoryWithStuff-> x ;
28         int w = pStuff->publicUnprotectedMemoryWithStuff-> y ;
29
30         printf( "%d %d", v, w ) ;
31     }
32 }
```

Theres a few assumptions here. One is that the pointer location used to "broadcast" the updates to the fields x, and y, resides in appropriately aligned memory (for us that means 8 byte aligned since we only support 64-bit systems now). Another assumption is that this alignment is sufficient that a store to the pointer (and subsequent access) wont ever be attempted in a fragmented way. We used to see on old HP PARISC systems generated assembly code where a 32-bit access was done with two 16-bit operations, so two halves of a pointer load or store wouldnt have a guarantee of being coherent. Given aligned memory on modern systems (ie: not PARISC) it isnt too unreasonable to assume that loads and stores are not done in this piecewise fashion, either by the compiler or the hardware.

Another assumption made here is that the compiler has generated code that executes in the same order as the higher level C code. Thats probably a bad assumption unless something else is done to explicitly enforce this ordering. On some systems we have intrinsics or low level compiler methods to enforce this ordering. One example is the no-op barrier mechanism available in the GCC compiler suite (or the Intel compiler that also implements GCC style inline assembly). So a more correct, but now platform specific, way of coding this broadcast function would be:

```
1 void updateMemoryAndLetEverybodySeeIt( broadcast * pStuff, sharedMem * pShared )
2 {
3     pStuff->x = 3 ;
4     pStuff->y = 7 ;
5
6     __asm__ __volatile__ ( "" ::: "memory" ) ;
7
8     pShared->publicUnprotectedMemoryWithStuff = pStuff ;
9 }
```

The next assumption here is one of sequential memory, one that also cannot be correctly presumed. By the time the publicUnprotectedMemoryWithStuff assignment is made, it is assumed that the previous x, and y memory operations are complete. i.e.: when somebody accesses $*pStuff->publicUnprotectedMemoryWithStuff$, the 3 and 7 values will be seen, and not some previous values. This is not correct on a system where out of order memory accesses are possible. Two example systems like this are AIX and HP-IPF (powerpc and ia64 respectively).

This code would actually work on HP-IPF since the compiler emits an st8.rel instruction for the volatile store. The .rel indicates that the store is to have "release" semantics, and behaves in a similar way to what is done in a mutex release operation: all previous loads and stores have to be complete before the effects of the store is visible to other cpus. That does exactly what the caller expects, by virtue of using volatile for the pointer. This requires that the caller know explicitly that volatile behaves this way on ia64 systems, unless one has explicitly disabled this behavior with suitable compilation options (every ia64 compiler Ive seen documents this volatile behavior).

However, on AIX (PowerPC), or Linux PPC, this code is not correct. There we need an lwsync instruction between the $pStuff->y$ store and the assignment of the pointer. Without that the store to publicUnprotectedMemoryWithStuff may occur before the stores to x, y are done. At the read point we have a data dependency that protects us, since we cant dereference without the pointer being non-null, so no value for x or y can be prefetched before the pointer is accessed and observed to be non-null. That wouldnt be the case if a flag (such a separate piece of atomically manipulated memory was used to flag the availability of the producers pair of stores. Again, we have platform specific requirements to make this code right. On Linux PPC, using the GCC compiler, one could do this with:

```
1 void updateMemoryAndLetEverybodySeeIt( broadcast * pStuff, sharedMem * pShared )
```

```
2 {
3     pStuff->x = 3 ;
4     pStuff->y = 7 ;
5
6     __asm__ __volatile__ ( "lwsync" ::: "memory" ) ;
7
8     pShared->publicUnprotectedMemoryWithStuff = pStuff ;
9 }
```

whereas on AIX, using xlC, one could use:

```
1 void updateMemoryAndLetEverybodySeeIt( broadcast * pStuff, sharedMem * pShared )
2 {
3     pStuff->x = 3 ;
4     pStuff->y = 7 ;
5
6     __lwsync() ; // from builtins.h
7
8     pShared->publicUnprotectedMemoryWithStuff = pStuff ;
9 }
```

The __lwsync() intrinsic also has the side effect of preventing code motion, so it does both the jobs of making the compiler generate the desired "sequential" code, and also makes the hardware do things in the same enforced ordered fashion. This sort of platform dependenence is the cost that you are forced to pay if you choose to attempt to avoid the use of mutex operations that would normally be used to make this code safe and correct.