Memory access ordering part 2 - barriers and the Linux kernel

Posted by **ARM Leif**,
11 April 2011

My previous post provided an introduction to the concept of memory access ordering. It did not however provide any solution to the problem, or necessarily specify where such ordering can be significant.

Now, not all software developers need to be deeply aware of memory access ordering or *barriers*. Unless your code interacts directly with hardware, interacts directly with code executing on other cores or directly loads or generates instructions to be executed, things will mostly Just Work. If your interaction with hardware is completely through a device driver (meaning: no device control registers mapped directly into your application), then it is the responsibility of the driver to enforce ordering. If your communication with software running on a different core makes use of a multithreading API, for example using Pthreads or Java threads, then it is the responsibility of that API to enforce ordering. If your program executes on an operating system that implements demand paging, then clearly it is the responsibility of the operating system to enforce ordering of such operations.

However, if you are writing device drivers, implementing your own thread-communications or creating a JIT compiler, then not being aware of the proper use of barriers can lead to unexpected and difficult to diagnose problems. Where your program requires a specific order of memory accesses to be seen by multiple cores or devices in the system, the solution is called barriers.

While the underlying architectural concepts are interesting in themselves, they are not what the majority of software developers concerned with barriers need to know about. For that reason, this post covers barrier use within the Linux kernel only. I promise to return to the gritty detail in a later post.

So, what are these barriers then?

A barrier, in some architectures called a *fence*, is an operation that explicitly enforces some type of ordering of memory accesses. On the higher level this can mean compiler directives preventing load/store operations from being reordered across a line in the source code, but leaving the compiler free to rearrange memory accesses on either side with other accesses on the same side. On the lower level, this can mean dedicated instructions stopping execution on a core until all previous memory accesses are guaranteed to be visible to other agents in the system. An agent is any device in the system capable of initiating bus transactions - for example a processor or a DMA controller.

Figure 1 shows an example of a barrier affecting the ordering of load-store instructions.

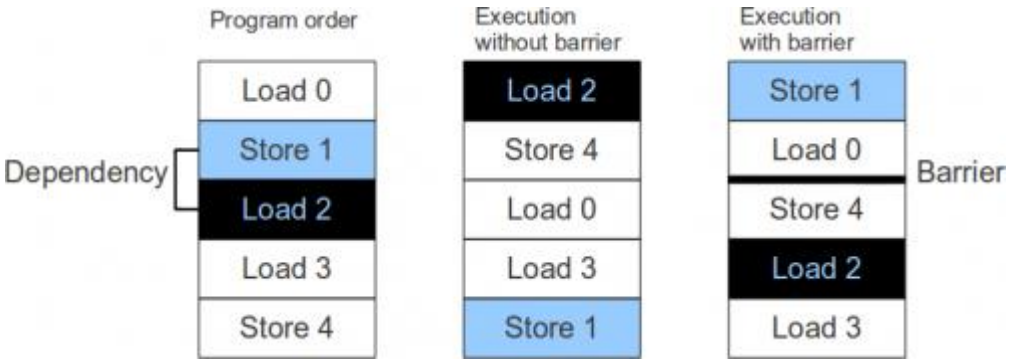

Figure 1: Example of barrier effect

There is an ordering dependency that the effects of Store 1 are visible to Load 2. For example, Store 1 might be a write to a configuration register that remaps the physical address of a peripheral that is then read from by Load 2. Note that accesses on either side of the barrier can still be freely reordered where there are no address dependencies.

Barriers in the Linux kernel

Because the compiler directives, barrier instructions and other system operations will differ between vendors, architectures and overall set of system components, the Linux kernel defines a portable set of barrier operations that need to be implemented for each architecture. Since the supported architecture with the weakest memory model (effectively the one that permits the most reordering) was the DEC Alpha, this was used as the reference architecture. No other architectures have since surpassed the DEC Alpha in this regard, but ARMv7-A comes pretty close. The full documentation of the barriers available in the Linux kernel can be found in linux/Documentation/memory-barriers.txt, but I will give a quick intro here.

Linux barrier API

General barrier

A general barrier has no runtime effect, it is only an instruction to the compiler to prevent reordering of memory accesses for optimization purposes.

Statement	Description
`barrier()`	Compiler barrier only. The compiler will not reorder memory accesses from one side of this statement to the other. This has no effect on the order that the processor actually executes the generated instructions.

Mandatory barriers

Mandatory barriers are used to enforce memory consistency on a full system level. The most common example of this is when communicating with external memory mapped peripherals. All mandatory barriers are guaranteed to expand to at least a compiler barrier, regardless of target architecture.

Statement	Description
`mb()`	A full system memory barrier. All memory operations before the mb() in the instruction stream will be committed before any operations after the mb() are committed. This ordering will be visible to all bus masters in the system. It will also ensure the order in which accesses from a single processor reaches slave devices.
`rmb()`	Like mb(), but only guarantees ordering between read accesses. That is, all read operations before an rmb() will be committed before any read operations after the rmb().
`wmb()`	Like mb(), but only guarantees ordering between write accesses. That is, all write operations before a wmb() will be committed before any write operations after the wmb().

SMP conditional barriers

The SMP conditional barriers are used to ensure a consistent view of memory between different cores within a cache coherent SMP system. When compiling a kernel without CONFIG_SMP, all SMP barriers are converted into plain compiler barriers.

Note: SMP barriers are a subset of mandatory barriers, not a superset (which is a common misunderstanding). An SMP barrier cannot replace a mandatory barrier, but a mandatory barrier can replace an SMP barrier.

Statement	Description
`smp_mb()`	Similar to mb(), but only guarantees ordering between cores/processors within an SMP system. All memory accesses before the smp_mb() will be visible to all cores within the SMP system before any accesses after the smp_mb().
`smp_rmb()`	Like smp_mb(), but only guarantees ordering between read accesses.
`smp_wmb()`	Like smp_mb(), but only guarantees ordering between write accesses.

Implicit barriers

Locking constructs available within the kernel act as implicit SMP barriers, in the same way as pthread synchronization operations do in user space. When using these to protect a shared resource, explicit barriers need not be used as well (for the purpose of ensuring consistency of that resource). This does not however remove the need for explicit barriers when communicating with external masters.

Due to a large number of device drivers not using the required barriers, I/O accessor macros (`readb()`, `iowrite32()` etcetera) for the ARM architecture act as explicit memory barriers when the kernel is compiled with CONFIG_ARM_DMA_MEM_BUFFERABLE. This was added in linux-2.6.35.

Other barriers

There are other barriers available within the Linux kernel as well. This post covered only the most commonly required ones. Please see the Linux kernel documentation for more information.

Usage examples

The Linux kernel patch submission guidelines state that *"All memory barriers {e.g., barrier(), rmb(), wmb()} need a comment in the source code that explains the logic of what they are doing and why."*. Although this is not always adhered to, this means that the kernel source itself can be a useful reference for the use of barriers. For example, the following is taken from `linux/drivers/net/8139too.c`:

```
        /*
               * Writing to TxStatus triggers a DMA transfer of the data
               * copied to tp->tx_buf[entry] above. Use a memory barrier
               * to make sure that the device sees the updated data.
               */
        wmb();
        RTL_W32_F (TxStatus0 + (entry * sizeof (u32)),
                               tp->tx_flag | max(len, (unsigned int)ETH_ZLEN));
```

This code is executed after some data has been written into a buffer to be handed over to a DMA engine. The `wmb()` ensures that the write into the buffer is committed before the write that initiates the DMA transaction, removing a risk for data corruption. Since only the ordering between these two specific accesses is necessary, and they are both writes, a `wmb()` is the correct choice. Note that this barrier is also SMP safe, as its description is a superset of the `smp_wmb()` functionality.

Another example from `linux/drivers/net/bnx2.c`:

```
        /* Memory barrier necessary as speculative reads of the rx
               * buffer can be ahead of the index in the status block
               */
        rmb();
        while (sw_cons != hw_cons) {
```

As described by the comment, in this situation the primary purpose of the barrier is to prevent the processor (as well as the compiler) from performing read accesses described within the `while` loop before that control block is actually entered.

Cost of barriers

The very reason for using barriers is to prevent our tools and hardware from performing unsafe optimizations. Also, the different types of barriers exist in order to describe exactly which memory ordering you need to enforce. This means that starting to insert barriers everywhere, or to use `mb()` wherever a barrier is needed, can have a negative impact on your software performance. It can be well worth spending the extra time to figure out whether you actually need a barrier in a specific situation, and if so which specific barrier it should be.

Future posts

My next post will be about the barrier instructions and operations available in the ARM architecture.

References

linux/Documentation/memory-barriers.txt
Paul E. McKenney - Memory Barriers: a Hardware View for Software Hackers

Leif, Principal Software Engineer, ARM, he first started playing around with UNIX in 1994, installing Linux in 1995, in 1999 graduated to Linux on embedded systems, and moved to doing it for a living in 2001. After four years of that, he packed up and moved to Cambridge, UK, to work for ARM. He is currently working on improving ARM multi-core support in (mostly) open source software. He is way too fond of blue LEDs (but any color will do if they are flashing), and is cowardly refusing to pick sides in the Linux/BSD war.