Memory barrier

From Wikipedia, the free encyclopedia

Memory barrier, also known as **membar** or **memory fence**, is a class of instructions which cause a central processing unit (CPU) to enforce an ordering constraint on memory operations issued before and after the barrier instruction.

CPUs employ performance optimizations that can result in out-of-order execution, including memory load and store operations. Memory operation reordering normally goes unnoticed within a single thread of execution, but causes unpredictable behaviour in concurrent programs and device drivers unless carefully controlled. The exact nature of an ordering constraint is hardware dependent, and defined by the architecture's memory model. Some architectures provide multiple barriers for enforcing different ordering constraints.

Memory barriers are typically used when implementing low-level machine code that operates on memory shared by multiple devices. Such code includes synchronization primitives and lock-free data structures on multiprocessor systems, and device drivers that communicate with computer hardware.

Contents

An illustrative example

When a program runs on a single CPU, the hardware performs the necessary book-keeping to ensure that programs execute as if all memory operations were performed in program order, hence memory barriers are not necessary. However, when the memory is shared with multiple devices, such as other CPUs in a multiprocessor system, or memory mapped peripherals, out-of-order access may affect program behavior. For example a second CPU may see memory changes made by the first CPU in a sequence which differs from program order.

The following two processor program gives a concrete example of how such out-of-order execution can affect program behavior:

Initially, memory locations x and f both hold the value 0. The program running on processor #1 loops until the value of f is non-zero, then it prints the value of x. The program running on processor #2 stores the value 42 into x and then stores the value 1 into f. Pseudo code for the two

program fragments is shown below. The steps of the program correspond to individual processor instructions.

```
Processor #1:
 loop:
  load the value in location f, if it is 0 goto loop
 print the value in location x

Processor #2:
 store the value 42 into location x
 store the value 1 into location f
```

You might expect the print statement to always print the number "42"; however, if processor #2's store operations are executed out-of-order, it is possible that f would be updated *before* x, and the print statement might print "0". For most programs this situation is not acceptable. A memory barrier can be inserted before processor #2's assignment to f to ensure that the new value of x was visible to other processors at or prior to the change in the value of f.

Low-level architecture-specific primitives

Memory barriers are low-level primitives which are part of the definition of an architecture's memory model. Like instruction sets, memory models vary considerably between architectures, so it is not appropriate to generalize about memory barrier behavior. The conventional wisdom is that using memory barriers correctly requires careful study of the architecture manuals for the hardware one is programming. That said, the following paragraph offers a glimpse of some memory barriers which exist in the wild.

Some architectures provide only a single memory barrier instruction sometimes called "full fence". A full fence ensures that all load and store operations prior to the fence will have been committed prior to any loads and stores issued following the fence. Other architectures provide separate "acquire" and "release" memory barriers which address the visibility of read-after-write operations from the point of view of a reader (sink) or writer (source) respectively. Some architectures provide separate memory barriers to control ordering between different combinations of system memory and I/O memory. When more than one memory barrier instruction is available it is important to consider that the cost of different instructions may vary considerably.

Multithreaded programming and memory visibility

See also: Memory model (computing)

Multithreaded programs usually use synchronisation primitives provided by a high-level programming environment, such as Java, or an API such as POSIX pthreads or Win32. Primitives such as mutexes and semaphores are provided to synchronize access to resources from parallel threads of execution. These primitives are usually implemented with the memory barriers required to provide the expected memory visibility semantics. In such environments explicit use of memory barriers is not generally necessary.

Each API or programming environment in principle has its own high-level memory model that defines its memory visibility semantics. Although programmers do not usually need to use memory barriers in such high level environments, it is important to understand their memory

visibility semantics, to the extent possible. Such understanding is not necessarily easy to achieve because memory visibility semantics are not always consistently specified or documented.

Just as programming language semantics are defined at a different level of abstraction to machine language opcodes, a programming environment's memory model is defined at a different level of abstraction to that of a hardware memory model. It is important to understand this distinction and realize that there is not always a simple mapping between low-level hardware memory barrier semantics and the high-level memory visibility semantics of a particular programming environment. As a result, a particular platform's implementation of (say) pthreads may employ stronger barriers than required by the specification. Programs which take advantage of memory visibility as-implemented rather than as-specified may not be portable.

Out-of-order execution versus compiler reordering optimizations

Memory barrier instructions only address reordering effects at the hardware level. Compilers may also reorder instructions as part of the program optimization process. Although the effects on parallel program behavior can be similar in both cases, in general it is necessary to take separate measures to inhibit compiler reordering optimizations for data that may be shared by multiple threads of execution. Note that such measures are usually only necessary for data which is not protected by synchronization primitives such as those discussed in the previous section.

In C and C++, the *volatile* keyword was intended to allow C and C++ programs to directly access Memory-mapped I/O. Memory-mapped I/O generally requires that the reads and writes specified in source code happen in the exact order specified in source code with no omissions. Omissions or reorderings of reads and writes by the compiler would break the communication between the program and the device accessed by Memory-mapped I/O. A C or C++ compiler may not reorder reads and writes to volatile memory locations, nor may it omit a read or write to a volatile memory location, allowing a pointer to volatile memory to be used for Memory-mapped I/O.

The C and C++ standards do not address multiple threads (or multiple processors), and as such, the usefulness of volatile depends on the compiler and hardware. Although volatile guarantees that the reads and writes will happen in the exact order specified in the source code, the compiler may generate code which reorders a volatile read or write with non-volatile reads or writes, thus limiting its usefulness as a inter-thread flag or mutex. Moreover, you are not guaranteed that volatile reads and writes will be seen in the same order by other processors due to caching, meaning volatile variables may not even work as inter-thread flags or mutexes.

Some languages and compilers may provide sufficient facilities to implement functions which address both the compiler reordering and machine reordering issues. In Java version 1.5 (also known as version 5), the *volatile* keyword is now guaranteed to prevent certain hardware *and* compiler re-orderings, as part of the new Java Memory Model. The proposed C++ memory model does not use *volatile*, instead C++0x will include special atomic types and operations with semantics similar to those of *volatile* in the Java Memory Model.

See also

- Lock-free and wait-free algorithms

External links

- Microsoft Driver Development: Memory Barriers on Multiprocessor Architectures (http://www.microsoft.com/whdc/driver/kernel/MPmem-barrier.mspx)
- HP technical report HPL-2004-209: Threads Cannot be Implemented as a Library (http://www.hpl.hp.com/techreports/2004/HPL-2004-209.html)
- Linux kernel memory barrier issues on multiple types of CPUs (http://www.linuxjournal.com/article/8211)
- Documentation on memory barriers in the Linux kernel (http://lxr.linux.no/source/Documentation/memory-barriers.txt)

- OldNewThing Blog on the subject (http://blogs.msdn.com/oldnewthing/archive/2004/05/28/143769.aspx)

Retrieved from "http://en.wikipedia.org/wiki/Memory_barrier"
Categories: Computer memory | Instruction processing
