ORACLE
SOLARIS

An Oracle White Paper
September 2010

Handling Memory Ordering in Multithreaded Applications with Oracle Solaris Studio 12 Update 2:
Part 2, Memory Barriers and Memory Fences

ORACLE

Introduction

Oracle Solaris Studio 12 Update 2 introduces a new header file `<mbarrier.h>`. This header file provides a set of memory ordering intrinsics that are useful for enforcing memory ordering constraints in multithreaded applications.

This article is part 2 of a two-part series. Part 1 discussed how compiler barriers can be used to stop the compiler from generating code that is incorrect due to reordered memory accesses. This part discusses how memory barriers or memory fences can be used to ensure that the processor does not reorder memory operations.

What Is Memory Ordering?

Memory ordering is the order in which memory operations (loads and stores) are performed. There are two ways that memory ordering might be changed:

- The compiler might change memory ordering as a result of some compile-time optimizations.
- The processor might change memory ordering at run time.

More About Memory Ordering

Memory ordering controls the order in which memory operations in one thread become visible to other threads running on the system. In single-threaded applications and in most multithreaded code, the order in which operations are committed is not important. However, there are situations where it is important that memory operations are visible to other processors in the same order in which the original processor executed them.

An example of when this is an issue is in the acquisition and release of mutex locks. Listing 1 shows a snippet of code in which a variable is modified and then a mutex lock is released.

It is important that the store operation to the variable `value` is completed before the store operation to release the lock is issued. If these two stores are interchanged by the processor at run time, then other processors will see the lock released before they see the new value of the variable `value` protected by the lock. Consequently, another processor might immediately acquire the lock and receive the old contents of the variable `value`.

Listing 1. Critical Section and Mutex Release

```
...
  LOAD  [&value], %o0  // Load value
  ADD   %o0, 1, %o0    // Increment
  STORE %o0, [&value]  // Store value
  STORE 0, [&lock]     // Release lock
```

To avoid this problem, we need to ensure that there is a memory barrier between the two store operations. This memory barrier ensures that hardware does not reorder the stores at run time. The modified code is shown in Listing 2.

Listing 2. Critical Section with Memory Barrier and Mutex Release

```
...
  LOAD  [&value], %o0  // Load value
  ADD   %o0, 1, %o0    // Increment
  STORE %o0, [&value]  // Store value
  MEMORYBARRIER        // Ensure that the stores remain ordered
  STORE 0, [&lock]     // Release lock
```

Strong and Weak Memory Ordering

Different processor types, different processor versions, and even one processor executing in different modes might have different policies towards enforcing memory ordering. SPARC and x86 processors implement what is known as strong memory ordering. With strong memory ordering, there are few situations in which it is necessary to explicitly include memory barriers.

Other processors might implement weak memory ordering. For those processors, it is necessary to include memory barriers in most situations where the order in which memory operations become visible is critical for the correct functioning of the application.

SPARC and x86 Memory Ordering

SPARC processors implement what is known as total store ordering (TSO), which ensures that store memory operations are visible to the rest of the system in the order in which they occurred.

x86 processors implement essentially the same memory ordering model. The *UltraSPARC Architecture 2007* manual defines the TSO behavior as follows:

"Loads are ordered with respect to earlier loads. Stores are ordered with respect to earlier loads and stores. Thus, loads can bypass earlier stores but cannot bypass earlier loads; stores cannot bypass earlier loads or stores."

The following diagrams show these constraints. Figure 1 illustrates the situation for streams of loads or stores. Each load operation cannot pass either earlier or later load operations, and the stream of load operations is ordered. The same constraint is true for store operations.

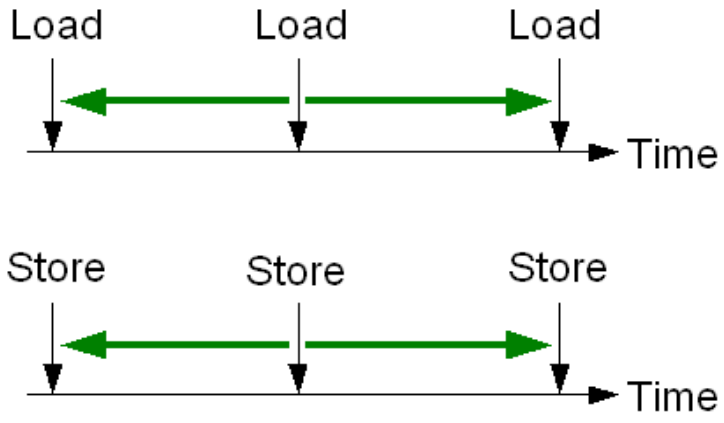

Figure 1. Ordering Constraints on Streams of Loads or Store Operations to Different Addresses

Figure 2 illustrates a more complex situation with mixed streams of load and store operations. In this situation, store operations can become visible after the execution of loads to different addresses that follow them in the instruction stream. Equivalently, later loads can be executed before the results of previous stores to different addresses become visible.

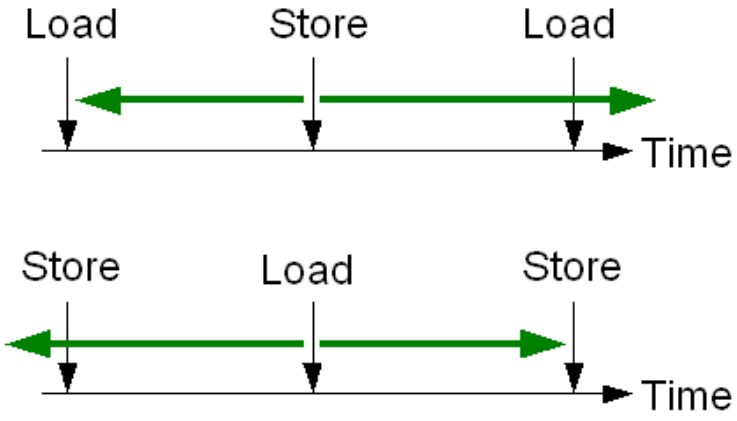

Figure 2. Ordering Constraints on Mixed Streams of Load and Store Operations to Different Addresses

The consequence of the implementation details on both SPARC and x86 processors is that the memory barrier shown in Listing 2 is not necessary. Although a memory barrier is not necessary for this particular code sequence, some other code sequences will require memory barriers to function correctly. For example, Dekker's Algorithm for mutual exclusion requires memory barriers to function correctly.

In some situations, it might be useful to include the memory barrier so the code will also run when ported to processors that implement a weaker memory ordering model. Processors that do not need the instructions ignore them.

Intrinsic Support for Memory Barriers

SPARC processors use a `membar` instruction to provide a memory barrier, and x86 processors provide an `mfence` instruction.

The particular variant of the instruction that is required depends on the situation in which the instruction is to be placed. Rather than expose this low level of detail to the user, the intrinsics defined in `<mbarrier.h>` provide a higher level of abstraction.

The available intrinsics are summarized in Table 1.

TABLE 1. MEMORY BARRIER INTRINSICS PROVIDED BY <MBARRIER.H>

INTRINSIC	DESCRIPTION
`__machine_r_barrier()`	All previous loads must have completed before the next memory operation commences.
`__machine_w_barrier()`	All previous writes must have completed before the next memory operation commences.
`__machine_rw_barrier()`	All previous memory operations must have completed before the next memory operation commences.
`__machine_acq_barrier()`	With the acquire memory barrier intrinsic, all previous memory operations must have completed before any future read operations commence.
`__machine_rel_barrier()`	With the release memory barrier intrinsic, all previous writes must have completed before any future memory operations commence.

References

- Page 8 through 14 of *Intel 64 and IA-32 Architectures Software Developer's Manual*: http://www.intel.com/Assets/PDF/manual/253668.pdf
- Page 93 of the *UltraSPARC Architecture 2007* manual: http://opensparc-t2.sunsource.net/specs/UA2007-current-draft-HP-EXT.pdf

Conclusion

This article, which is part 2 of a two-part series, described how memory barriers or memory fences can be used to ensure that a processor does not reorder memory operations.

Part 1 of the series described how to use compiler barriers to stop the compiler from generating code that is incorrect due to reordered memory accesses.

Oracle is committed to developing practices and products that help protect the environment

Handling Memory Ordering in Multithreaded Applications with Oracle Solaris Studio 12 Update 2: Part 2, Memory Barriers and Memory Fences
September 2010
Author: Darryl Gove

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com