**Memory Barriers**

Memory barriers are used to provide control over the order of memory accesses. This is necessary sometimes because optimizations performed by the compiler and hardware can cause memory to be accessed in a different order than intended by the developer.

A memory barrier affects instructions that access memory in two ways:

* provides control over the order that memory access instructions are performed, and
* provides control over when memory access instructions will complete.

Memory access instructions, such as loads and stores, typically take longer to execute than other instructions. Therefore, compilers use registers to hold frequently used values and processors use high speed caches to hold the most frequently used memory locations. Another common optimization is for compilers and processors to rearrange the order that instructions are executed so that the processor does not have to wait for memory accesses to complete. This can result in memory being accessed in a different order than specified in the source code. While this typically will not cause a problem in a single thread of execution, it can cause a problem if the location can also be accessed from another processor or device.

**Types of Memory Barriers**

As mentioned above, both compilers and processors can optimize the execution of instructions in a way that necessitates the use of a memory barrier. A memory barrier that affects both the compiler and the processor is a *hardware memory barrier*, and a memory barrier that only affects the compiler is a *software memory barrier*.

In addition to hardware and software memory barriers, a memory barrier can be restricted to memory reads, memory writes, or both. A memory barrier that affects both reads and writes is a *full memory barrier*.

There is also a class of memory barrier that is specific to multi-processor environments. The name of these memory barriers are prefixed with "smp". On a multi-processor system, these barriers are hardware memory barriers and on uni-processor systems, they are software memory barriers.

The barrier() macro is the only software memory barrier, and it is a full memory barrier. All other memory barriers in the Linux kernel are hardware barriers. A hardware memory barrier is an implied software barrier.

**Using Memory Barriers**

The two most common needs for memory barriers are to manage memory shared by more than one processor and IO control registers that are mapped to memory locations.

In the case of shared memory, when there is only one CPU, hardware memory barriers are not needed. Because of this, memory barriers that are only needed to control shared memory between processors can be optimized for better performance on systems with only one processor. As mentioned above, the name of these memory barriers are prefixed with "smp".

**Memory Barrier Interfaces**

**mb()**

#include <asm/system.h>

void mb(void);

This function inserts a hardware memory barrier that prevents any memory access from being moved and executed on the other side of the barrier. It guarantees that any memory access initiated before the memory barrier will be complete before passing the barrier, and all subsequent memory accesses will be executed after the barrier.

**rmb()**

#include <asm/system.h>

void rmb(void);

This function inserts a hardware memory barrier that prevents any memory read access from being moved and executed on the other side of the barrier. It guarantees that any memory read access initiated before the memory barrier will be complete before passing the barrier, and all subsequent memory read accesses will be executed after the barrier.

**wmb()**

#include <asm/system.h>

void wmb(void);

This function inserts a hardware memory barrier that prevents any memory write access from being moved and executed on the other side of the barrier. It guarantees that any memory write access initiated before the memory barrier will be complete before passing the barrier, and all subsequent memory write accesses will be executed after the barrier.

**barrier()**

#include <linux/kernel.h>

void barrier(void);

This function inserts a software memory barrier that affects the compiler code generation, but it does not affect the hardware's execution of instructions. The compiler will save to memory any modified values that it has loaded in registers, and it will reread all values from memory the next time they are needed.

**smp\_mb()**

#include <asm/system.h>

void smp\_mb(void);

This function is the same as the mb() function on multi-processor systems, and it is the same as the barrier() function on uni-processor systems.

**smp\_rmb()**

#include <asm/system.h>

void smp\_rmb(void);

This function is the same as the rmb() function on multi-processor systems, and it is the same as the barrier() function on uni-processor systems.

**smp\_wmb()**

#include <asm/system.h>

void smp\_wmb(void);

This function is the same as the wmb() function on multi-processor systems, and it is the same as the barrier() function on uni-processor systems.