LINUX KERNEL MEMORY BARRIERS

By: David Howells <dhowells@redhat.com>
    Paul E. McKenney <paulmck@linux.vnet.ibm.com>

Contents:

    - Cache coherency vs DMA.
    - Cache coherency vs MMIO.

 (*) The things CPUs get up to.

    - And then there's the Alpha.

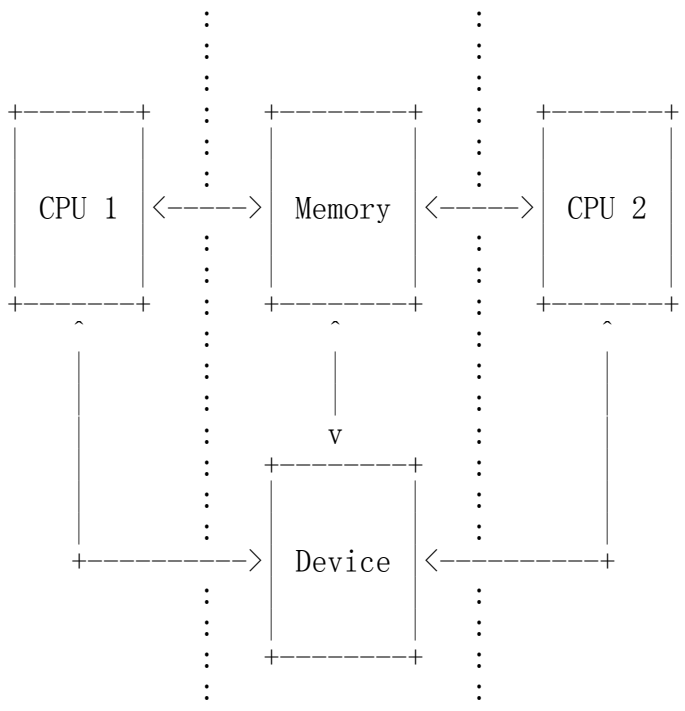
 (*) Example uses.

    - Circular buffers.

 (*) References.

============================
ABSTRACT MEMORY ACCESS MODEL
============================

Consider the following abstract model of the system:



Each CPU executes a program that generates memory access operations. In the abstract CPU, memory operation ordering is very relaxed, and a CPU may actually perform the memory operations in any order it likes, provided program causality appears to be maintained. Similarly, the compiler may also arrange the instructions it emits in any order it likes, provided it doesn't affect the apparent operation of the program.

So in the above diagram, the effects of the memory operations performed by a CPU are perceived by the rest of the system as the operations cross the interface between the CPU and rest of the system (the dotted lines).

For example, consider the following sequence of events:

```
CPU 1           CPU 2
=============== ===============
{ A == 1; B == 2 }
A = 3;          x = A;
B = 4;          y = B;
```

The set of accesses as seen by the memory system in the middle can be arranged in 24 different combinations:

```
STORE A=3,      STORE B=4,      x=LOAD A->3,    y=LOAD B->4
STORE A=3,      STORE B=4,      y=LOAD B->4,    x=LOAD A->3
STORE A=3,      x=LOAD A->3,    STORE B=4,      y=LOAD B->4
STORE A=3,      x=LOAD A->3,    y=LOAD B->2,    STORE B=4
STORE A=3,      y=LOAD B->2,    STORE B=4,      x=LOAD A->3
STORE A=3,      y=LOAD B->2,    x=LOAD A->3,    STORE B=4
STORE B=4,      STORE A=3,      x=LOAD A->3,    y=LOAD B->4
STORE B=4, ...
...
```

and can thus result in four different combinations of values:

```
x == 1, y == 2
x == 1, y == 4
x == 3, y == 2
x == 3, y == 4
```

Furthermore, the stores committed by a CPU to the memory system may not be perceived by the loads made by another CPU in the same order as the stores were committed.

As a further example, consider this sequence of events:

```
CPU 1           CPU 2
=============== ===============
{ A == 1, B == 2, C = 3, P == &A, Q == &C }
B = 4;          Q = P;
P = &B          D = *Q;
```

There is an obvious data dependency here, as the value loaded into D depends on the address retrieved from P by CPU 2. At the end of the sequence, any of the following results are possible:

```
(Q == &A) and (D == 1)
(Q == &B) and (D == 2)
(Q == &B) and (D == 4)
```

Note that CPU 2 will never try and load C into D because the CPU will load P into Q before issuing the load of *Q.

DEVICE OPERATIONS

Some devices present their control interfaces as collections of memory locations, but the order in which the control registers are accessed is very important. For instance, imagine an ethernet card with a set of internal registers that are accessed through an address port register (A) and a data port register (D). To read internal register 5, the following code might then be used:

```
*A = 5;
x = *D;
```

but this might show up as either of the following two sequences:

```
STORE *A = 5, x = LOAD *D
x = LOAD *D, STORE *A = 5
```

the second of which will almost certainly result in a malfunction, since it set the address _after_ attempting to read the register.

GUARANTEES

There are some minimal guarantees that may be expected of a CPU:

(*) On any given CPU, dependent memory accesses will be issued in order, with respect to itself. This means that for:

```
Q = P; D = *Q;
```

the CPU will issue the following memory operations:

```
Q = LOAD P, D = LOAD *Q
```

and always in that order.

(*) Overlapping loads and stores within a particular CPU will appear to be ordered within that CPU. This means that for:

```
a = *X; *X = b;
```

the CPU will only issue the following sequence of memory operations:

```
a = LOAD *X, STORE *X = b
```

And for:

```
*X = c; d = *X;
```

the CPU will only issue:

```
STORE *X = c, d = LOAD *X
```

(Loads and stores overlap if they are targeted at overlapping pieces of memory).

And there are a number of things that _must_ or _must_not_ be assumed:

(*) It _must_not_ be assumed that independent loads and stores will be issued
    in the order given. This means that for:

        X = *A; Y = *B; *D = Z;

    we may get any of the following sequences:

        X = LOAD *A, Y = LOAD *B, STORE *D = Z
        X = LOAD *A, STORE *D = Z, Y = LOAD *B
        Y = LOAD *B, X = LOAD *A, STORE *D = Z
        Y = LOAD *B, STORE *D = Z, X = LOAD *A
        STORE *D = Z, X = LOAD *A, Y = LOAD *B
        STORE *D = Z, Y = LOAD *B, X = LOAD *A

(*) It _must_ be assumed that overlapping memory accesses may be merged or
    discarded. This means that for:

        X = *A; Y = *(A + 4);

    we may get any one of the following sequences:

        X = LOAD *A; Y = LOAD *(A + 4);
        Y = LOAD *(A + 4); X = LOAD *A;
        {X, Y} = LOAD {*A, *(A + 4) };

    And for:

        *A = X; Y = *A;

    we may get either of:

        STORE *A = X; Y = LOAD *A;
        STORE *A = Y = X;

=========================
WHAT ARE MEMORY BARRIERS?
=========================

As can be seen above, independent memory operations are effectively performed
in random order, but this can be a problem for CPU-CPU interaction and for I/O.
What is required is some way of intervening to instruct the compiler and the
CPU to restrict the order.

Memory barriers are such interventions. They impose a perceived partial
ordering over the memory operations on either side of the barrier.

Such enforcement is important because the CPUs and other devices in a system
can use a variety of tricks to improve performance, including reordering,
deferral and combination of memory operations; speculative loads; speculative
branch prediction and various types of caching. Memory barriers are used to
override or suppress these tricks, allowing the code to sanely control the
interaction of multiple CPUs and/or devices.

VARIETIES OF MEMORY BARRIER

Memory barriers come in four basic varieties:

(1) Write (or store) memory barriers.

A write memory barrier gives a guarantee that all the STORE operations specified before the barrier will appear to happen before all the STORE operations specified after the barrier with respect to the other components of the system.

A write barrier is a partial ordering on stores only; it is not required to have any effect on loads.

A CPU can be viewed as committing a sequence of store operations to the memory system as time progresses. All stores before a write barrier will occur in the sequence _before_ all the stores after the write barrier.

[!] Note that write barriers should normally be paired with read or data dependency barriers; see the "SMP barrier pairing" subsection.

(2) Data dependency barriers.

A data dependency barrier is a weaker form of read barrier. In the case where two loads are performed such that the second depends on the result of the first (eg: the first load retrieves the address to which the second load will be directed), a data dependency barrier would be required to make sure that the target of the second load is updated before the address obtained by the first load is accessed.

A data dependency barrier is a partial ordering on interdependent loads only; it is not required to have any effect on stores, independent loads or overlapping loads.

As mentioned in (1), the other CPUs in the system can be viewed as committing sequences of stores to the memory system that the CPU being considered can then perceive. A data dependency barrier issued by the CPU under consideration guarantees that for any load preceding it, if that load touches one of a sequence of stores from another CPU, then by the time the barrier completes, the effects of all the stores prior to that touched by the load will be perceptible to any loads issued after the data dependency barrier.

See the "Examples of memory barrier sequences" subsection for diagrams showing the ordering constraints.

[!] Note that the first load really has to have a _data_ dependency and not a control dependency. If the address for the second load is dependent on the first load, but the dependency is through a conditional rather than actually loading the address itself, then it's a _control_ dependency and a full read barrier or better is required. See the "Control dependencies" subsection for more information.

[!] Note that data dependency barriers should normally be paired with

write barriers; see the "SMP barrier pairing" subsection.

(3) Read (or load) memory barriers.

A read barrier is a data dependency barrier plus a guarantee that all the LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier with respect to the other components of the system.

A read barrier is a partial ordering on loads only; it is not required to have any effect on stores.

Read memory barriers imply data dependency barriers, and so can substitute for them.

[!] Note that read barriers should normally be paired with write barriers; see the "SMP barrier pairing" subsection.

(4) General memory barriers.

A general memory barrier gives a guarantee that all the LOAD and STORE operations specified before the barrier will appear to happen before all the LOAD and STORE operations specified after the barrier with respect to the other components of the system.

A general memory barrier is a partial ordering over both loads and stores.

General memory barriers imply both read and write memory barriers, and so can substitute for either.

And a couple of implicit varieties:

(5) LOCK operations.

This acts as a one-way permeable barrier. It guarantees that all memory operations after the LOCK operation will appear to happen after the LOCK operation with respect to the other components of the system.

Memory operations that occur before a LOCK operation may appear to happen after it completes.

A LOCK operation should almost always be paired with an UNLOCK operation.

(6) UNLOCK operations.

This also acts as a one-way permeable barrier. It guarantees that all memory operations before the UNLOCK operation will appear to happen before the UNLOCK operation with respect to the other components of the system.

Memory operations that occur after an UNLOCK operation may appear to happen before it completes.

LOCK and UNLOCK operations are guaranteed to appear with respect to each other strictly in the order specified.

The use of LOCK and UNLOCK operations generally precludes the need for other sorts of memory barrier (but note the exceptions mentioned in the subsection "MMIO write barrier").

Memory barriers are only required where there's a possibility of interaction between two CPUs or between a CPU and a device. If it can be guaranteed that there won't be any such interaction in any particular piece of code, then memory barriers are unnecessary in that piece of code.

Note that these are the _minimum_ guarantees. Different architectures may give more substantial guarantees, but they may _not_ be relied upon outside of arch specific code.

WHAT MAY NOT BE ASSUMED ABOUT MEMORY BARRIERS?

There are certain things that the Linux kernel memory barriers do not guarantee:

(*) There is no guarantee that any of the memory accesses specified before a memory barrier will be _complete_ by the completion of a memory barrier instruction; the barrier can be considered to draw a line in that CPU's access queue that accesses of the appropriate type may not cross.

(*) There is no guarantee that issuing a memory barrier on one CPU will have any direct effect on another CPU or any other hardware in the system. The indirect effect will be the order in which the second CPU sees the effects of the first CPU's accesses occur, but see the next point:

(*) There is no guarantee that a CPU will see the correct order of effects from a second CPU's accesses, even _if_ the second CPU uses a memory barrier, unless the first CPU _also_ uses a matching memory barrier (see the subsection on "SMP Barrier Pairing").

(*) There is no guarantee that some intervening piece of off-the-CPU hardware[*] will not reorder the memory accesses. CPU cache coherency mechanisms should propagate the indirect effects of a memory barrier between CPUs, but might not do so in order.

[*] For information on bus mastering DMA and coherency please read:

Documentation/PCI/pci.txt
Documentation/PCI/PCI-DMA-mapping.txt
Documentation/DMA-API.txt

DATA DEPENDENCY BARRIERS

The usage requirements of data dependency barriers are a little subtle, and it's not always obvious that they're needed. To illustrate, consider the

following sequence of events:

```
        CPU 1           CPU 2
        =============== ===============
        { A == 1, B == 2, C = 3, P == &A, Q == &C }
        B = 4;
        <write barrier>
        P = &B
                        Q = P;
                        D = *Q;
```

There's a clear data dependency here, and it would seem that by the end of the sequence, Q must be either &A or &B, and that:

```
        (Q == &A) implies (D == 1)
        (Q == &B) implies (D == 4)
```

But! CPU 2's perception of P may be updated _before_ its perception of B, thus leading to the following situation:

```
        (Q == &B) and (D == 2) ????
```

Whilst this may seem like a failure of coherency or causality maintenance, it isn't, and this behaviour can be observed on certain real CPUs (such as the DEC Alpha).

To deal with this, a data dependency barrier or better must be inserted between the address load and the data load:

```
        CPU 1           CPU 2
        =============== ===============
        { A == 1, B == 2, C = 3, P == &A, Q == &C }
        B = 4;
        <write barrier>
        P = &B
                        Q = P;
                        <data dependency barrier>
                        D = *Q;
```

This enforces the occurrence of one of the two implications, and prevents the third possibility from arising.

[!] Note that this extremely counterintuitive situation arises most easily on machines with split caches, so that, for example, one cache bank processes even-numbered cache lines and the other bank processes odd-numbered cache lines. The pointer P might be stored in an odd-numbered cache line, and the variable B might be stored in an even-numbered cache line. Then, if the even-numbered bank of the reading CPU's cache is extremely busy while the odd-numbered bank is idle, one can see the new value of the pointer P (&B), but the old value of the variable B (2).

Another example of where data dependency barriers might by required is where a number is read from memory and then used to calculate the index for an array access:

```
CPU 1                 CPU 2
===============       ===============
{ M[0] == 1, M[1] == 2, M[3] = 3, P == 0, Q == 3 }
M[1] = 4;
<write barrier>
P = 1
                      Q = P;
                      <data dependency barrier>
                      D = M[Q];
```

The data dependency barrier is very important to the RCU system, for example. See rcu_dereference() in include/linux/rcupdate.h. This permits the current target of an RCU'd pointer to be replaced with a new modified target, without the replacement target appearing to be incompletely initialised.

See also the subsection on "Cache Coherency" for a more thorough example.

CONTROL DEPENDENCIES

A control dependency requires a full read memory barrier, not simply a data dependency barrier to make it work correctly. Consider the following bit of code:

```
q = &a;
if (p)
        q = &b;
<data dependency barrier>
x = *q;
```

This will not have the desired effect because there is no actual data dependency, but rather a control dependency that the CPU may short-circuit by attempting to predict the outcome in advance. In such a case what's actually required is:

```
q = &a;
if (p)
        q = &b;
<read barrier>
x = *q;
```

SMP BARRIER PAIRING

When dealing with CPU-CPU interactions, certain types of memory barrier should always be paired. A lack of appropriate pairing is almost certainly an error.

A write barrier should always be paired with a data dependency barrier or read barrier, though a general barrier would also be viable. Similarly a read barrier or a data dependency barrier should always be paired with at least an write barrier, though, again, a general barrier is viable:

```
CPU 1                 CPU 2
```

```
               ============== ==============
               a = 1;
               <write barrier>
               b = 2;         x = b;
                              <read barrier>
                              y = a;
```

Or:

```
               CPU 1          CPU 2
               ============== ================================
               a = 1;
               <write barrier>
               b = &a;        x = b;
                              <data dependency barrier>
                              y = *x;
```

Basically, the read barrier always has to be there, even though it can be of the "weaker" type.

[!] Note that the stores before the write barrier would normally be expected to match the loads after the read barrier or the data dependency barrier, and vice versa:

```
               CPU 1                           CPU 2
               ==============                  ==============
               a = 1;           }----   --->{  v = c
               b = 2;           }    \ /    {  w = d
               <write barrier>          \      <read barrier>
               c = 3;           }    / \    {  x = a;
               d = 4;           }----   --->{  y = b;
```

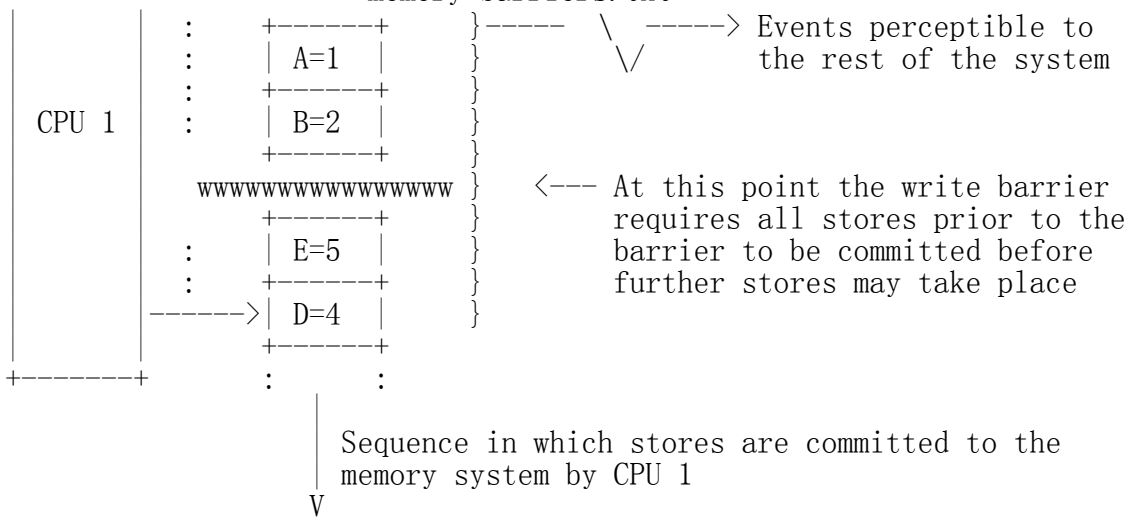
EXAMPLES OF MEMORY BARRIER SEQUENCES

Firstly, write barriers act as partial orderings on store operations. Consider the following sequence of events:

```
               CPU 1
               ======================
               STORE A = 1
               STORE B = 2
               STORE C = 3
               <write barrier>
               STORE D = 4
               STORE E = 5
```

This sequence of events is committed to the memory coherence system in an order that the rest of the system might perceive as the unordered set of { STORE A, STORE B, STORE C } all occurring before the unordered set of { STORE D, STORE E }:

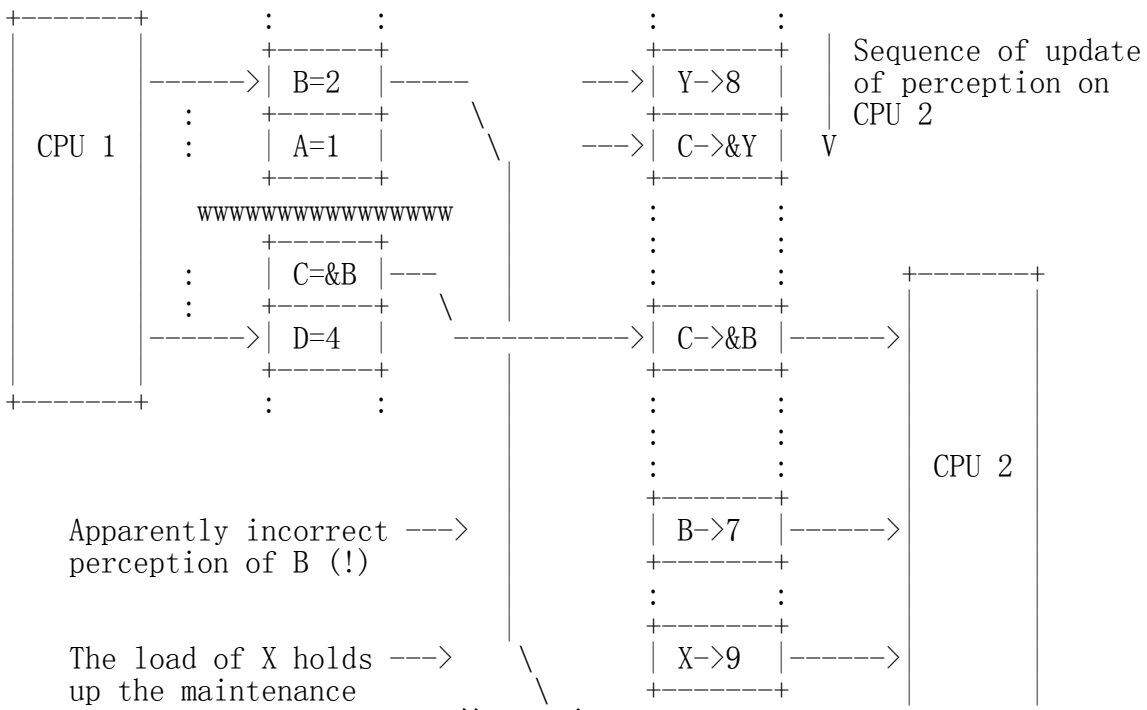
```
               +-------+       :      :
               |       |       +------+
               |       |------>| C=3  |     }     /\
```

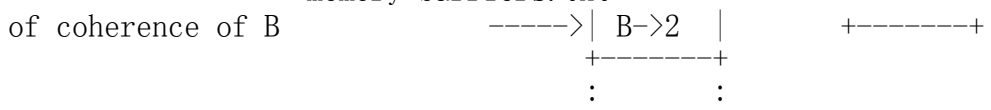


Secondly, data dependency barriers act as partial orderings on data-dependent loads. Consider the following sequence of events:

```
CPU 1                  CPU 2
====================== ======================
        { B = 7; X = 9; Y = 8; C = &Y }
STORE A = 1
STORE B = 2
<write barrier>
STORE C = &B           LOAD X
STORE D = 4            LOAD C (gets &B)
                       LOAD *C (reads B)
```

Without intervention, CPU 2 may perceive the events on CPU 1 in some effectively random order, despite the write barrier issued by CPU 1:



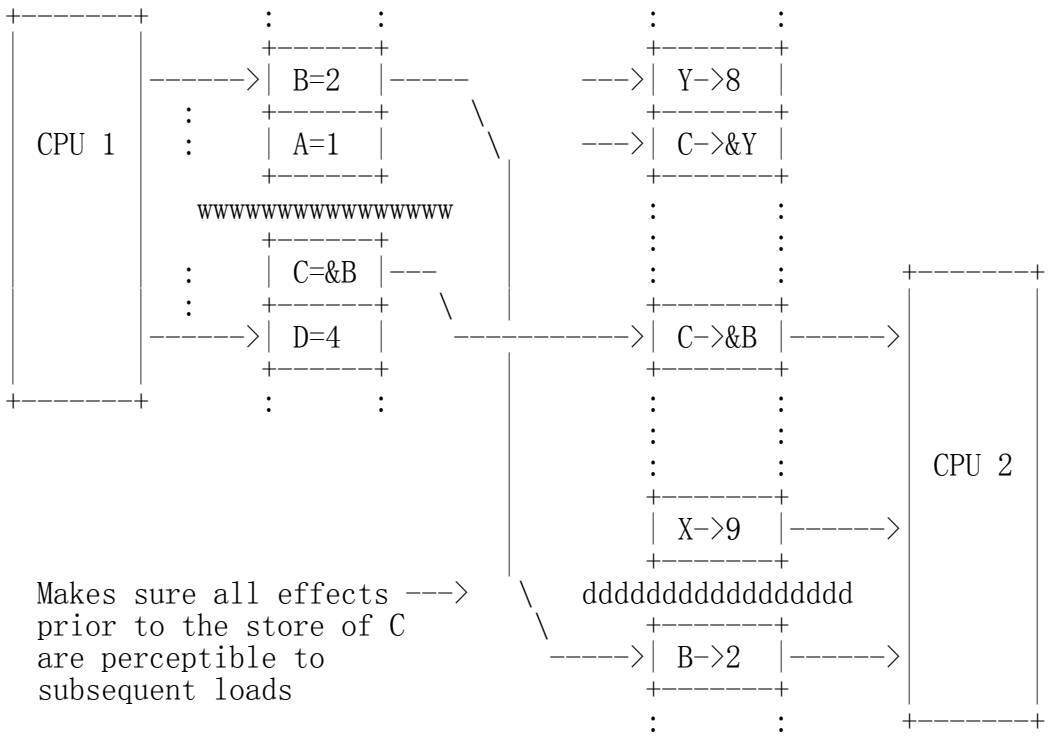


In the above example, CPU 2 perceives that B is 7, despite the load of *C (which would be B) coming after the LOAD of C.

If, however, a data dependency barrier were to be placed between the load of C and the load of *C (ie: B) on CPU 2:

```
CPU 1                   CPU 2
======================= =======================
        { B = 7; X = 9; Y = 8; C = &Y }
STORE A = 1
STORE B = 2
<write barrier>
STORE C = &B            LOAD X
STORE D = 4             LOAD C (gets &B)
                        <data dependency barrier>
                        LOAD *C (reads B)
```

then the following will occur:

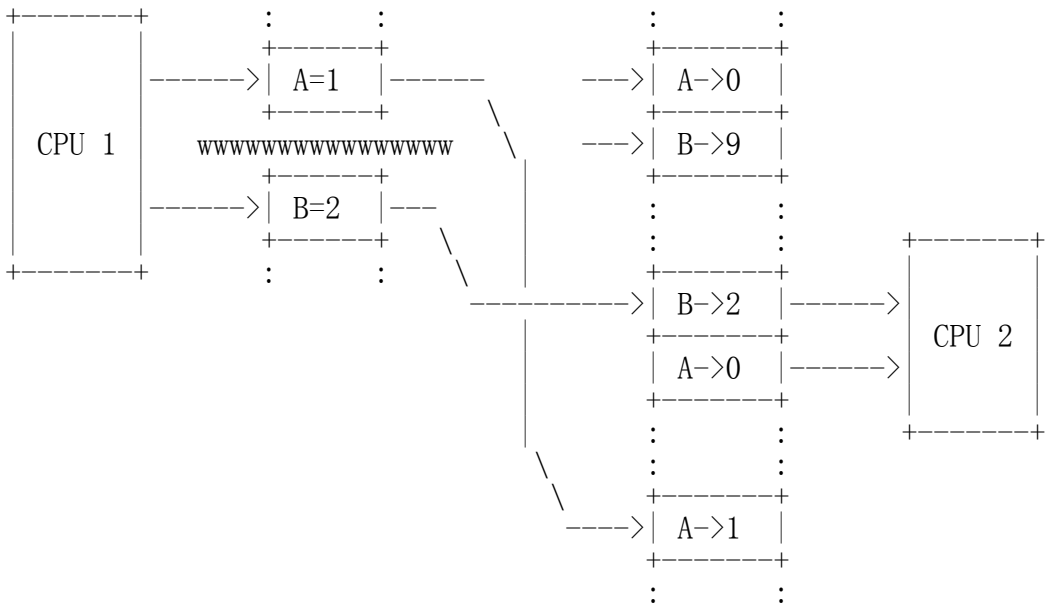


And thirdly, a read barrier acts as a partial order on loads. Consider the following sequence of events:

```
CPU 1                   CPU 2
======================= =======================
        { A = 0, B = 9 }
STORE A=1
```

```
<write barrier>
STORE B=2
                        LOAD B
                        LOAD A
```

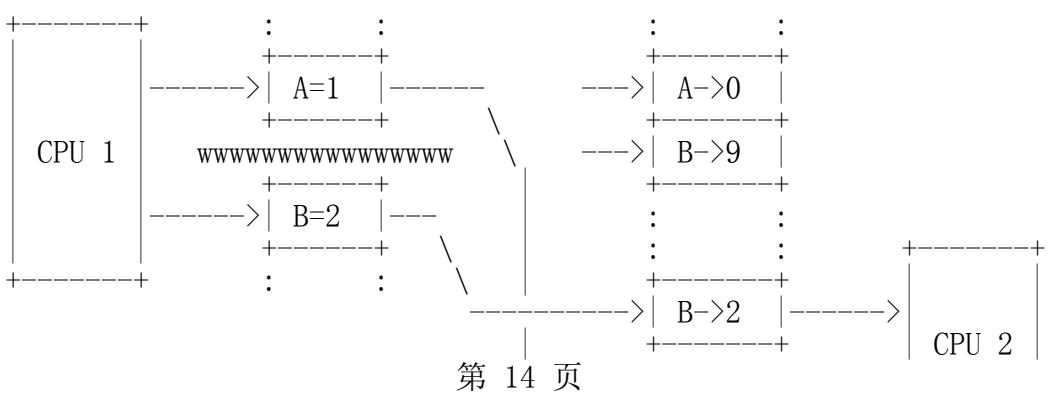
Without intervention, CPU 2 may then choose to perceive the events on CPU 1 in some effectively random order, despite the write barrier issued by CPU 1:

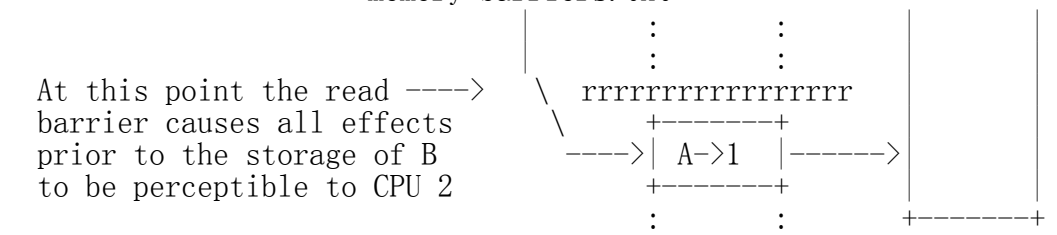


If, however, a read barrier were to be placed between the load of B and the load of A on CPU 2:

```
CPU 1                   CPU 2
======================= =======================
        { A = 0, B = 9 }
STORE A=1
<write barrier>
STORE B=2
                        LOAD B
                        <read barrier>
                        LOAD A
```

then the partial ordering imposed by CPU 1 will be perceived correctly by CPU 2:

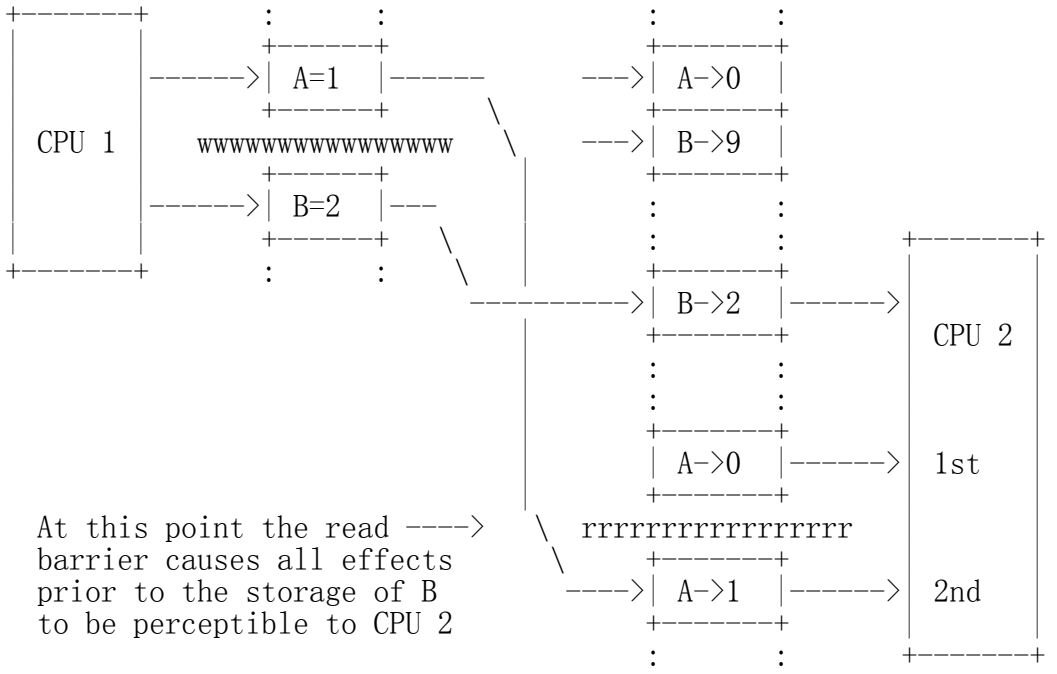




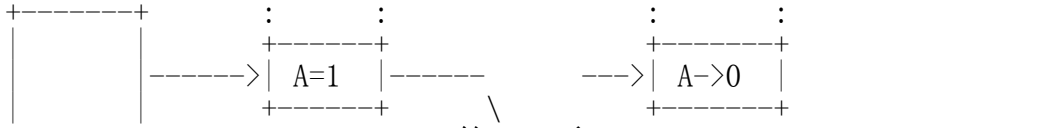
To illustrate this more completely, consider what could happen if the code contained a load of A either side of the read barrier:

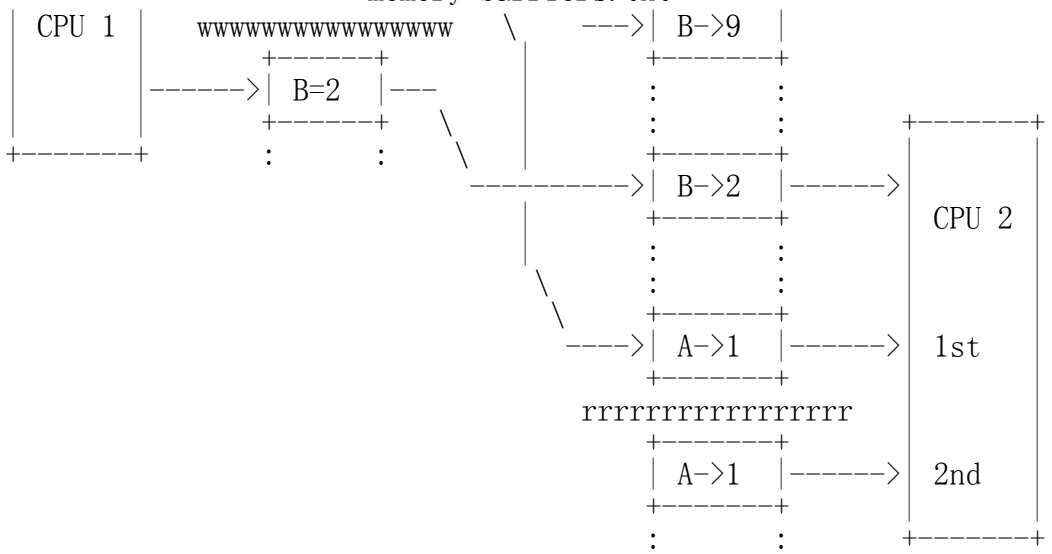
```
CPU 1                   CPU 2
======================= =======================
        { A = 0, B = 9 }
STORE A=1
<write barrier>
STORE B=2
                        LOAD B
                        LOAD A [first load of A]
                        <read barrier>
                        LOAD A [second load of A]
```

Even though the two loads of A both occur after the load of B, they may both come up with different values:



But it may be that the update to A from CPU 1 becomes perceptible to CPU 2 before the read barrier completes anyway:





The guarantee is that the second load will always come up with A == 1 if the load of B came up with B == 2. No such guarantee exists for the first load of A; that may come up with either A == 0 or A == 1.

READ MEMORY BARRIERS VS LOAD SPECULATION

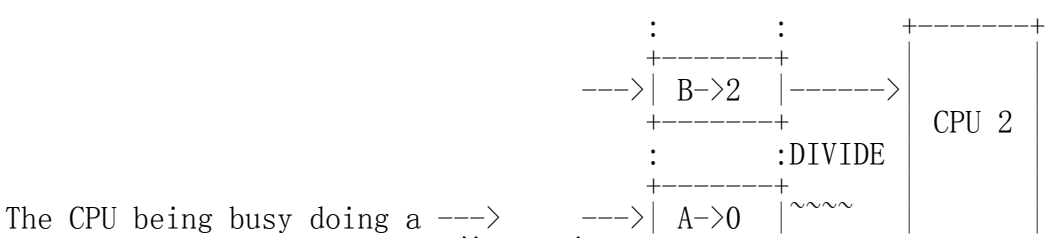
Many CPUs speculate with loads: that is they see that they will need to load an item from memory, and they find a time where they're not using the bus for any other loads, and so do the load in advance - even though they haven't actually got to that point in the instruction execution flow yet. This permits the actual load instruction to potentially complete immediately because the CPU already has the value to hand.

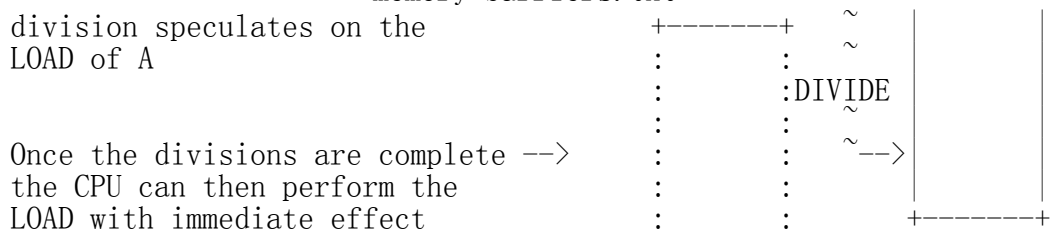
It may turn out that the CPU didn't actually need the value - perhaps because a branch circumvented the load - in which case it can discard the value or just cache it for later use.

Consider:

```
        CPU 1                   CPU 2
        ======================= =======================
                                LOAD B
                                DIVIDE          } Divide instructions generally
                                DIVIDE          } take a long time to perform
                                LOAD A
```

Which might appear as this:

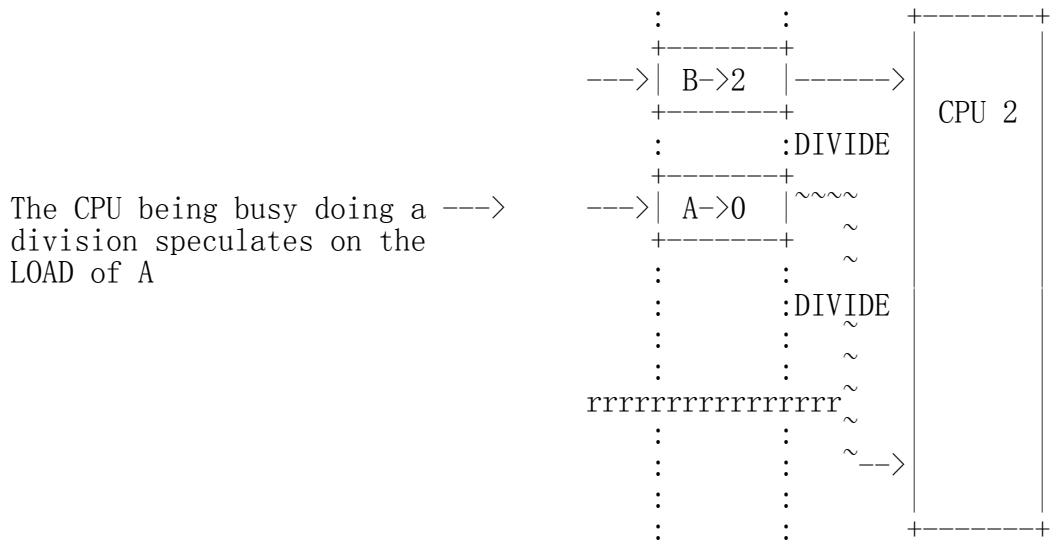




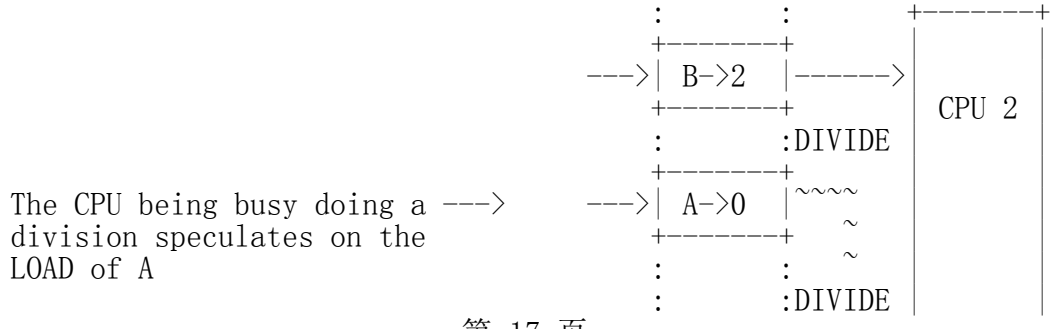
Placing a read barrier or a data dependency barrier just before the second load:

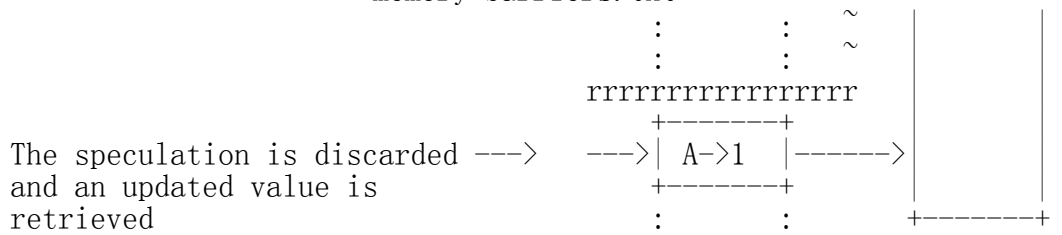
```
CPU 1                  CPU 2
====================== ======================
                       LOAD B
                       DIVIDE
                       DIVIDE
                       <read barrier>
                       LOAD A
```

will force any value speculatively obtained to be reconsidered to an extent dependent on the type of barrier used. If there was no change made to the speculated memory location, then the speculated value will just be used:



but if there was an update or an invalidation from another CPU pending, then the speculation will be cancelled and the value reloaded:





=======================
EXPLICIT KERNEL BARRIERS
=======================

The Linux kernel has a variety of different barriers that act at different levels:

  (*) Compiler barrier.

  (*) CPU memory barriers.

  (*) MMIO write barrier.

COMPILER BARRIER

The Linux kernel has an explicit compiler barrier function that prevents the compiler from moving the memory accesses either side of it to the other side:

```
barrier();
```

This is a general barrier - lesser varieties of compiler barrier do not exist.

The compiler barrier has no direct effect on the CPU, which may then reorder things however it wishes.

CPU MEMORY BARRIERS

The Linux kernel has eight basic CPU memory barriers:

TYPE	MANDATORY	SMP CONDITIONAL
GENERAL	mb()	smp_mb()
WRITE	wmb()	smp_wmb()
READ	rmb()	smp_rmb()
DATA DEPENDENCY	read_barrier_depends()	smp_read_barrier_depends()

All memory barriers except the data dependency barriers imply a compiler barrier. Data dependencies do not impose any additional compiler ordering.

Aside: In the case of data dependencies, the compiler would be expected to issue the loads in the correct order (eg. `a[b]` would have to load the value of b before loading a[b]), however there is no guarantee in the C specification

that the compiler may not speculate the value of b (eg. is equal to 1) and load a before b (eg. tmp = a[1]; if (b != 1) tmp = a[b];). There is also the problem of a compiler reloading b after having loaded a[b], thus having a newer copy of b than a[b]. A consensus has not yet been reached about these problems, however the ACCESS_ONCE macro is a good place to start looking.

SMP memory barriers are reduced to compiler barriers on uniprocessor compiled systems because it is assumed that a CPU will appear to be self-consistent, and will order overlapping accesses correctly with respect to itself.

[!] Note that SMP memory barriers _must_ be used to control the ordering of references to shared memory on SMP systems, though the use of locking instead is sufficient.

Mandatory barriers should not be used to control SMP effects, since mandatory barriers unnecessarily impose overhead on UP systems. They may, however, be used to control MMIO effects on accesses through relaxed memory I/O windows. These are required even on non-SMP systems as they affect the order in which memory operations appear to a device by prohibiting both the compiler and the CPU from reordering them.

There are some more advanced barrier functions:

(*) set_mb(var, value)

    This assigns the value to the variable and then inserts a full memory barrier after it, depending on the function. It isn't guaranteed to insert anything more than a compiler barrier in a UP compilation.

(*) smp_mb__before_atomic_dec();
(*) smp_mb__after_atomic_dec();
(*) smp_mb__before_atomic_inc();
(*) smp_mb__after_atomic_inc();

    These are for use with atomic add, subtract, increment and decrement functions that don't return a value, especially when used for reference counting. These functions do not imply memory barriers.

    As an example, consider a piece of code that marks an object as being dead and then decrements the object's reference count:

        obj->dead = 1;
        smp_mb__before_atomic_dec();
        atomic_dec(&obj->ref_count);

    This makes sure that the death mark on the object is perceived to be set *before* the reference counter is decremented.

    See Documentation/atomic_ops.txt for more information. See the "Atomic operations" subsection for information on where to use these.

(*) smp_mb__before_clear_bit(void);
(*) smp_mb__after_clear_bit(void);

These are for use similar to the atomic inc/dec barriers. These are typically used for bitwise unlocking operations, so care must be taken as there are no implicit memory barriers here either.

Consider implementing an unlock operation of some nature by clearing a locking bit. The clear_bit() would then need to be barriered like this:

```
smp_mb__before_clear_bit();
clear_bit( ... );
```

This prevents memory operations before the clear leaking to after it. See the subsection on "Locking Functions" with reference to UNLOCK operation implications.

See Documentation/atomic_ops.txt for more information. See the "Atomic operations" subsection for information on where to use these.

MMIO WRITE BARRIER

The Linux kernel also has a special barrier for use with memory-mapped I/O writes:

```
mmiowb();
```

This is a variation on the mandatory write barrier that causes writes to weakly ordered I/O regions to be partially ordered. Its effects may go beyond the CPU->Hardware interface and actually affect the hardware at some level.

See the subsection "Locks vs I/O accesses" for more information.

IMPLICIT KERNEL MEMORY BARRIERS

Some of the other functions in the linux kernel imply memory barriers, amongst which are locking and scheduling functions.

This specification is a _minimum_ guarantee; any particular architecture may provide more substantial guarantees, but these may not be relied upon outside of arch specific code.

LOCKING FUNCTIONS

The Linux kernel has a number of locking constructs:

(*) spin locks
(*) R/W spin locks
(*) mutexes
(*) semaphores
(*) R/W semaphores

 (*) RCU

In all cases there are variants on "LOCK" operations and "UNLOCK" operations
for each construct. These operations all imply certain barriers:

 (1) LOCK operation implication:

     Memory operations issued after the LOCK will be completed after the LOCK
     operation has completed.

     Memory operations issued before the LOCK may be completed after the LOCK
     operation has completed.

 (2) UNLOCK operation implication:

     Memory operations issued before the UNLOCK will be completed before the
     UNLOCK operation has completed.

     Memory operations issued after the UNLOCK may be completed before the
     UNLOCK operation has completed.

 (3) LOCK vs LOCK implication:

     All LOCK operations issued before another LOCK operation will be completed
     before that LOCK operation.

 (4) LOCK vs UNLOCK implication:

     All LOCK operations issued before an UNLOCK operation will be completed
     before the UNLOCK operation.

     All UNLOCK operations issued before a LOCK operation will be completed
     before the LOCK operation.

 (5) Failed conditional LOCK implication:

     Certain variants of the LOCK operation may fail, either due to being
     unable to get the lock immediately, or due to receiving an unblocked
     signal whilst asleep waiting for the lock to become available. Failed
     locks do not imply any sort of barrier.

Therefore, from (1), (2) and (4) an UNLOCK followed by an unconditional LOCK is
equivalent to a full barrier, but a LOCK followed by an UNLOCK is not.

[!] Note: one of the consequences of LOCKs and UNLOCKs being only one-way
    barriers is that the effects of instructions outside of a critical section
    may seep into the inside of the critical section.

A LOCK followed by an UNLOCK may not be assumed to be full memory barrier
because it is possible for an access preceding the LOCK to happen after the
LOCK, and an access following the UNLOCK to happen before the UNLOCK, and the
two accesses can themselves then cross:

        *A = a;
        LOCK
        UNLOCK

```
        *B = b;
```

may occur as:

```
        LOCK, STORE *B, STORE *A, UNLOCK
```

Locks and semaphores may not provide any guarantee of ordering on UP compiled systems, and so cannot be counted on in such a situation to actually achieve anything at all - especially with respect to I/O accesses - unless combined with interrupt disabling operations.

See also the section on "Inter-CPU locking barrier effects".

As an example, consider the following:

```
        *A = a;
        *B = b;
        LOCK
        *C = c;
        *D = d;
        UNLOCK
        *E = e;
        *F = f;
```

The following sequence of events is acceptable:

```
        LOCK, {*F,*A}, *E, {*C,*D}, *B, UNLOCK
```

        [+] Note that {*F,*A} indicates a combined access.

But none of the following are:

```
        {*F,*A}, *B,    LOCK, *C, *D,   UNLOCK, *E
        *A, *B, *C,     LOCK, *D,       UNLOCK, *E, *F
        *A, *B,         LOCK, *C,       UNLOCK, *D, *E, *F
        *B,             LOCK, *C, *D,   UNLOCK, {*F,*A}, *E
```

INTERRUPT DISABLING FUNCTIONS

Functions that disable interrupts (LOCK equivalent) and enable interrupts (UNLOCK equivalent) will act as compiler barriers only. So if memory or I/O barriers are required in such a situation, they must be provided from some other means.

SLEEP AND WAKE-UP FUNCTIONS

Sleeping and waking on an event flagged in global data can be viewed as an interaction between two pieces of data: the task state of the task waiting for the event and the global data used to indicate the event. To make sure that these appear to happen in the right order, the primitives to begin the process

of going to sleep, and the primitives to initiate a wake up imply certain barriers.

Firstly, the sleeper normally follows something like this sequence of events:

```
for (;;) {
        set_current_state(TASK_UNINTERRUPTIBLE);
        if (event_indicated)
                break;
        schedule();
}
```

A general memory barrier is interpolated automatically by set_current_state() after it has altered the task state:

```
CPU 1
===============================
set_current_state();
  set_mb();
    STORE current->state
    <general barrier>
LOAD event_indicated
```

set_current_state() may be wrapped by:

```
prepare_to_wait();
prepare_to_wait_exclusive();
```

which therefore also imply a general memory barrier after setting the state. The whole sequence above is available in various canned forms, all of which interpolate the memory barrier in the right place:

```
wait_event();
wait_event_interruptible();
wait_event_interruptible_exclusive();
wait_event_interruptible_timeout();
wait_event_killable();
wait_event_timeout();
wait_on_bit();
wait_on_bit_lock();
```

Secondly, code that performs a wake up normally follows something like this:

```
event_indicated = 1;
wake_up(&event_wait_queue);
```

or:

```
event_indicated = 1;
wake_up_process(event_daemon);
```

A write memory barrier is implied by wake_up() and co. if and only if they wake something up. The barrier occurs before the task state is cleared, and so sits between the STORE to indicate the event and the STORE to set TASK_RUNNING:

```
        CPU 1                           CPU 2
        =============================== ===============================
        set_current_state();            STORE event_indicated
          set_mb();                     wake_up();
            STORE current->state          <write barrier>
            <general barrier>             STORE current->state
        LOAD event_indicated
```

The available waker functions include:

```
        complete();
        wake_up();
        wake_up_all();
        wake_up_bit();
        wake_up_interruptible();
        wake_up_interruptible_all();
        wake_up_interruptible_nr();
        wake_up_interruptible_poll();
        wake_up_interruptible_sync();
        wake_up_interruptible_sync_poll();
        wake_up_locked();
        wake_up_locked_poll();
        wake_up_nr();
        wake_up_poll();
        wake_up_process();
```

[!] Note that the memory barriers implied by the sleeper and the waker do _not_ order multiple stores before the wake-up with respect to loads of those stored values after the sleeper has called set_current_state(). For instance, if the sleeper does:

```
        set_current_state(TASK_INTERRUPTIBLE);
        if (event_indicated)
                break;
        __set_current_state(TASK_RUNNING);
        do_something(my_data);
```

and the waker does:

```
        my_data = value;
        event_indicated = 1;
        wake_up(&event_wait_queue);
```

there's no guarantee that the change to event_indicated will be perceived by the sleeper as coming after the change to my_data. In such a circumstance, the code on both sides must interpolate its own memory barriers between the separate data accesses. Thus the above sleeper ought to do:

```
        set_current_state(TASK_INTERRUPTIBLE);
        if (event_indicated) {
                smp_rmb();
                do_something(my_data);
        }
```

and the waker should do:

```
	my_data = value;
	smp_wmb();
	event_indicated = 1;
	wake_up(&event_wait_queue);
```

MISCELLANEOUS FUNCTIONS

Other functions that imply barriers:

 (*) schedule() and similar imply full memory barriers.

==================================
INTER-CPU LOCKING BARRIER EFFECTS
==================================

On SMP systems locking primitives give a more substantial form of barrier: one that does affect memory access ordering on other CPUs, within the context of conflict on any particular lock.

LOCKS VS MEMORY ACCESSES

Consider the following: the system has a pair of spinlocks (M) and (Q), and three CPUs; then should the following sequence of events occur:

```
	CPU 1                           CPU 2
	=============================== ===============================
	*A = a;                         *E = e;
	LOCK M                          LOCK Q
	*B = b;                         *F = f;
	*C = c;                         *G = g;
	UNLOCK M                        UNLOCK Q
	*D = d;                         *H = h;
```

Then there is no guarantee as to what order CPU 3 will see the accesses to *A through *H occur in, other than the constraints imposed by the separate locks on the separate CPUs. It might, for example, see:

```
	*E, LOCK M, LOCK Q, *G, *C, *F, *A, *B, UNLOCK Q, *D, *H, UNLOCK M
```

But it won't see any of:

```
	*B, *C or *D preceding LOCK M
	*A, *B or *C following UNLOCK M
	*F, *G or *H preceding LOCK Q
	*E, *F or *G following UNLOCK Q
```

However, if the following occurs:

```
	CPU 1                           CPU 2
```

```
        ================================ ================================
        *A = a;
        LOCK M          [1]
        *B = b;
        *C = c;
        UNLOCK M        [1]
        *D = d;                          *E = e;
                                         LOCK M          [2]
                                         *F = f;
                                         *G = g;
                                         UNLOCK M        [2]
                                         *H = h;
```

CPU 3 might see:

```
        *E, LOCK M [1], *C, *B, *A, UNLOCK M [1],
                LOCK M [2], *H, *F, *G, UNLOCK M [2], *D
```

But assuming CPU 1 gets the lock first, CPU 3 won't see any of:

```
        *B, *C, *D, *F, *G or *H preceding LOCK M [1]
        *A, *B or *C following UNLOCK M [1]
        *F, *G or *H preceding LOCK M [2]
        *A, *B, *C, *E, *F or *G following UNLOCK M [2]
```

LOCKS VS I/O ACCESSES

Under certain circumstances (especially involving NUMA), I/O accesses within two spinlocked sections on two different CPUs may be seen as interleaved by the PCI bridge, because the PCI bridge does not necessarily participate in the cache-coherence protocol, and is therefore incapable of issuing the required read memory barriers.

For example:

```
        CPU 1                            CPU 2
        ================================ ================================
        spin_lock(Q)
        writel(0, ADDR)
        writel(1, DATA);
        spin_unlock(Q);
                                         spin_lock(Q);
                                         writel(4, ADDR);
                                         writel(5, DATA);
                                         spin_unlock(Q);
```

may be seen by the PCI bridge as follows:

```
        STORE *ADDR = 0, STORE *ADDR = 4, STORE *DATA = 1, STORE *DATA = 5
```

which would probably cause the hardware to malfunction.

What is necessary here is to intervene with an mmiowb() before dropping the

spinlock, for example:

```
        CPU 1                           CPU 2
        =============================== ===============================
        spin_lock(Q)
        writel(0, ADDR)
        writel(1, DATA);
        mmiowb();
        spin_unlock(Q);
                                        spin_lock(Q);
                                        writel(4, ADDR);
                                        writel(5, DATA);
                                        mmiowb();
                                        spin_unlock(Q);
```

this will ensure that the two stores issued on CPU 1 appear at the PCI bridge before either of the stores issued on CPU 2.

Furthermore, following a store by a load from the same device obviates the need for the mmiowb(), because the load forces the store to complete before the load is performed:

```
        CPU 1                           CPU 2
        =============================== ===============================
        spin_lock(Q)
        writel(0, ADDR)
        a = readl(DATA);
        spin_unlock(Q);
                                        spin_lock(Q);
                                        writel(4, ADDR);
                                        b = readl(DATA);
                                        spin_unlock(Q);
```

See Documentation/DocBook/deviceiobook.tmpl for more information.

WHERE ARE MEMORY BARRIERS NEEDED?

Under normal operation, memory operation reordering is generally not going to be a problem as a single-threaded linear piece of code will still appear to work correctly, even if it's in an SMP kernel. There are, however, four circumstances in which reordering definitely _could_ be a problem:

(*) Interprocessor interaction.

(*) Atomic operations.

(*) Accessing devices.

(*) Interrupts.

INTERPROCESSOR INTERACTION

When there's a system with more than one processor, more than one CPU in the system may be working on the same data set at the same time. This can cause synchronisation problems, and the usual way of dealing with them is to use locks. Locks, however, are quite expensive, and so it may be preferable to operate without the use of a lock if at all possible. In such a case operations that affect both CPUs may have to be carefully ordered to prevent a malfunction.

Consider, for example, the R/W semaphore slow path. Here a waiting process is queued on the semaphore, by virtue of it having a piece of its stack linked to the semaphore's list of waiting processes:

```
struct rw_semaphore {
        ...
        spinlock_t lock;
        struct list_head waiters;
};

struct rwsem_waiter {
        struct list_head list;
        struct task_struct *task;
};
```

To wake up a particular waiter, the up_read() or up_write() functions have to:

(1) read the next pointer from this waiter's record to know as to where the next waiter record is;

(2) read the pointer to the waiter's task structure;

(3) clear the task pointer to tell the waiter it has been given the semaphore;

(4) call wake_up_process() on the task; and

(5) release the reference held on the waiter's task struct.

In other words, it has to perform this sequence of events:

```
LOAD waiter->list.next;
LOAD waiter->task;
STORE waiter->task;
CALL wakeup
RELEASE task
```

and if any of these steps occur out of order, then the whole thing may malfunction.

Once it has queued itself and dropped the semaphore lock, the waiter does not get the lock again; it instead just waits for its task pointer to be cleared before proceeding. Since the record is on the waiter's stack, this means that if the task pointer is cleared _before_ the next pointer in the list is read, another CPU might start processing the waiter and might clobber the waiter's stack before the up*() function has a chance to read the next pointer.

Consider then what might happen to the above sequence of events:

```
        CPU 1                           CPU 2
        =============================== ===============================
                                        down_xxx()
                                        Queue waiter
                                        Sleep
        up_yyy()
        LOAD waiter->task;
        STORE waiter->task;
                                        Woken up by other event
        <preempt>
                                        Resume processing
                                        down_xxx() returns
                                        call foo()
                                        foo() clobbers *waiter
        </preempt>
        LOAD waiter->list.next;
        --- OOPS ---
```

This could be dealt with using the semaphore lock, but then the down_xxx() function has to needlessly get the spinlock again after being woken up.

The way to deal with this is to insert a general SMP memory barrier:

```
        LOAD waiter->list.next;
        LOAD waiter->task;
        smp_mb();
        STORE waiter->task;
        CALL wakeup
        RELEASE task
```

In this case, the barrier makes a guarantee that all memory accesses before the barrier will appear to happen before all the memory accesses after the barrier with respect to the other CPUs on the system. It does _not_ guarantee that all the memory accesses before the barrier will be complete by the time the barrier instruction itself is complete.

On a UP system - where this wouldn't be a problem - the smp_mb() is just a compiler barrier, thus making sure the compiler emits the instructions in the right order without actually intervening in the CPU. Since there's only one CPU, that CPU's dependency ordering logic will take care of everything else.

ATOMIC OPERATIONS

Whilst they are technically interprocessor interaction considerations, atomic operations are noted specially as some of them imply full memory barriers and some don't, but they're very heavily relied on as a group throughout the kernel.

Any atomic operation that modifies some state in memory and returns information about the state (old or new) implies an SMP-conditional general memory barrier (smp_mb()) on each side of the actual operation (with the exception of

explicit lock operations, described later). These include:

```
xchg();
cmpxchg();
atomic_cmpxchg();
atomic_inc_return();
atomic_dec_return();
atomic_add_return();
atomic_sub_return();
atomic_inc_and_test();
atomic_dec_and_test();
atomic_sub_and_test();
atomic_add_negative();
atomic_add_unless();    /* when succeeds (returns 1) */
test_and_set_bit();
test_and_clear_bit();
test_and_change_bit();
```

These are used for such things as implementing LOCK-class and UNLOCK-class operations and adjusting reference counters towards object destruction, and as such the implicit memory barrier effects are necessary.

The following operations are potential problems as they do _not_ imply memory barriers, but might be used for implementing such things as UNLOCK-class operations:

```
atomic_set();
set_bit();
clear_bit();
change_bit();
```

With these the appropriate explicit memory barrier should be used if necessary (smp_mb__before_clear_bit() for instance).

The following also do _not_ imply memory barriers, and so may require explicit memory barriers under some circumstances (smp_mb__before_atomic_dec() for instance):

```
atomic_add();
atomic_sub();
atomic_inc();
atomic_dec();
```

If they're used for statistics generation, then they probably don't need memory barriers, unless there's a coupling between statistical data.

If they're used for reference counting on an object to control its lifetime, they probably don't need memory barriers because either the reference count will be adjusted inside a locked section, or the caller will already hold sufficient references to make the lock, and thus a memory barrier unnecessary.

If they're used for constructing a lock of some description, then they probably do need memory barriers as a lock primitive generally has to do things in a specific order.

Basically, each usage case has to be carefully considered as to whether memory barriers are needed or not.

The following operations are special locking primitives:

```
test_and_set_bit_lock();
clear_bit_unlock();
__clear_bit_unlock();
```

These implement LOCK-class and UNLOCK-class operations. These should be used in preference to other operations when implementing locking primitives, because their implementations can be optimised on many architectures.

[!] Note that special memory barrier primitives are available for these situations because on some CPUs the atomic instructions used imply full memory barriers, and so barrier instructions are superfluous in conjunction with them, and in such cases the special barrier primitives will be no-ops.

See Documentation/atomic_ops.txt for more information.

ACCESSING DEVICES

Many devices can be memory mapped, and so appear to the CPU as if they're just a set of memory locations. To control such a device, the driver usually has to make the right memory accesses in exactly the right order.

However, having a clever CPU or a clever compiler creates a potential problem in that the carefully sequenced accesses in the driver code won't reach the device in the requisite order if the CPU or the compiler thinks it is more efficient to reorder, combine or merge accesses - something that would cause the device to malfunction.

Inside of the Linux kernel, I/O should be done through the appropriate accessor routines - such as inb() or writel() - which know how to make such accesses appropriately sequential. Whilst this, for the most part, renders the explicit use of memory barriers unnecessary, there are a couple of situations where they might be needed:

(1) On some systems, I/O stores are not strongly ordered across all CPUs, and so for _all_ general drivers locks should be used and mmiowb() must be issued prior to unlocking the critical section.

(2) If the accessor functions are used to refer to an I/O memory window with relaxed memory access properties, then _mandatory_ memory barriers are required to enforce ordering.

See Documentation/DocBook/deviceiobook.tmpl for more information.

INTERRUPTS

A driver may be interrupted by its own interrupt service routine, and thus the

two parts of the driver may interfere with each other's attempts to control or access the device.

This may be alleviated - at least in part - by disabling local interrupts (a form of locking), such that the critical operations are all contained within the interrupt-disabled section in the driver. Whilst the driver's interrupt routine is executing, the driver's core may not run on the same CPU, and its interrupt is not permitted to happen again until the current interrupt has been handled, thus the interrupt handler does not need to lock against that.

However, consider a driver that was talking to an ethernet card that sports an address register and a data register. If that driver's core talks to the card under interrupt-disablement and then the driver's interrupt handler is invoked:

```
LOCAL IRQ DISABLE
writew(ADDR, 3);
writew(DATA, y);
LOCAL IRQ ENABLE
<interrupt>
writew(ADDR, 4);
q = readw(DATA);
</interrupt>
```

The store to the data register might happen after the second store to the address register if ordering rules are sufficiently relaxed:

```
STORE *ADDR = 3, STORE *ADDR = 4, STORE *DATA = y, q = LOAD *DATA
```

If ordering rules are relaxed, it must be assumed that accesses done inside an interrupt disabled section may leak outside of it and may interleave with accesses performed in an interrupt - and vice versa - unless implicit or explicit barriers are used.

Normally this won't be a problem because the I/O accesses done inside such sections will include synchronous load operations on strictly ordered I/O registers that form implicit I/O barriers. If this isn't sufficient then an mmiowb() may need to be used explicitly.

A similar situation may occur between an interrupt routine and two routines running on separate CPUs that communicate with each other. If such a case is likely, then interrupt-disabling locks should be used to guarantee ordering.

```
==========================
KERNEL I/O BARRIER EFFECTS
==========================
```

When accessing I/O memory, drivers should use the appropriate accessor functions:

(*) inX(), outX():

    These are intended to talk to I/O space rather than memory space, but that's primarily a CPU-specific concept. The i386 and x86_64 processors do

indeed have special I/O space access cycles and instructions, but many CPUs don't have such a concept.

The PCI bus, amongst others, defines an I/O space concept which - on such CPUs as i386 and x86_64 - readily maps to the CPU's concept of I/O space. However, it may also be mapped as a virtual I/O space in the CPU's memory map, particularly on those CPUs that don't support alternate I/O spaces.

Accesses to this space may be fully synchronous (as on i386), but intermediary bridges (such as the PCI host bridge) may not fully honour that.

They are guaranteed to be fully ordered with respect to each other.

They are not guaranteed to be fully ordered with respect to other types of memory and I/O operation.

(*) readX(), writeX():

Whether these are guaranteed to be fully ordered and uncombined with respect to each other on the issuing CPU depends on the characteristics defined for the memory window through which they're accessing. On later i386 architecture machines, for example, this is controlled by way of the MTRR registers.

Ordinarily, these will be guaranteed to be fully ordered and uncombined, provided they're not accessing a prefetchable device.

However, intermediary hardware (such as a PCI bridge) may indulge in deferral if it so wishes; to flush a store, a load from the same location is preferred[*], but a load from the same device or from configuration space should suffice for PCI.

[*] NOTE! attempting to load from the same location as was written to may cause a malfunction - consider the 16550 Rx/Tx serial registers for example.

Used with prefetchable I/O memory, an mmiowb() barrier may be required to force stores to be ordered.

Please refer to the PCI specification for more information on interactions between PCI transactions.

(*) readX_relaxed()

These are similar to readX(), but are not guaranteed to be ordered in any way. Be aware that there is no I/O read barrier available.

(*) ioreadX(), iowriteX()

These will perform appropriately for the type of access they're actually doing, be it inX()/outX() or readX()/writeX().

ASSUMED MINIMUM EXECUTION ORDERING MODEL
==

It has to be assumed that the conceptual CPU is weakly-ordered but that it will maintain the appearance of program causality with respect to itself. Some CPUs (such as i386 or x86_64) are more constrained than others (such as powerpc or frv), and so the most relaxed case (namely DEC Alpha) must be assumed outside of arch-specific code.

This means that it must be considered that the CPU will execute its instruction stream in any order it feels like - or even in parallel - provided that if an instruction in the stream depends on an earlier instruction, then that earlier instruction must be sufficiently complete[*] before the later instruction may proceed; in other words: provided that the appearance of causality is maintained.

[*] Some instructions have more than one effect - such as changing the condition codes, changing registers or changing memory - and different instructions may depend on different effects.

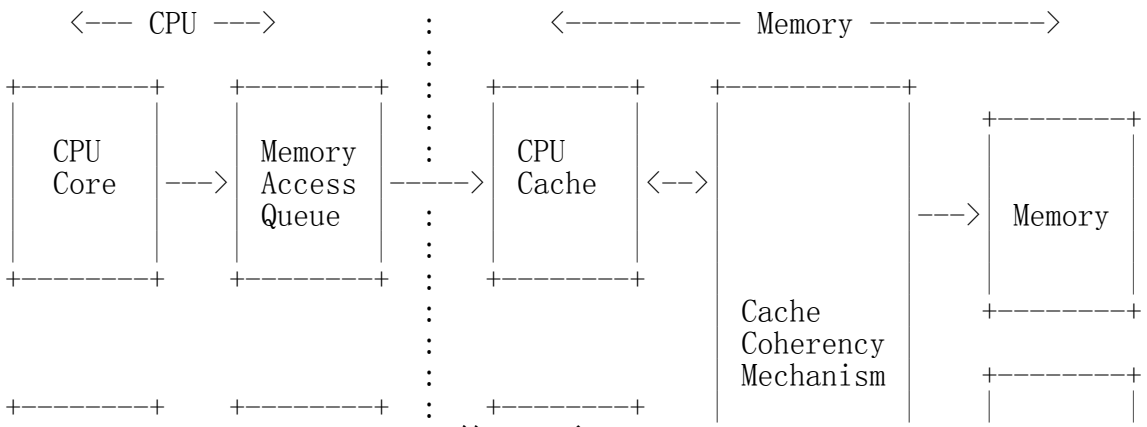
A CPU may also discard any instruction sequence that winds up having no ultimate effect. For example, if two adjacent instructions both load an immediate value into the same register, the first may be discarded.

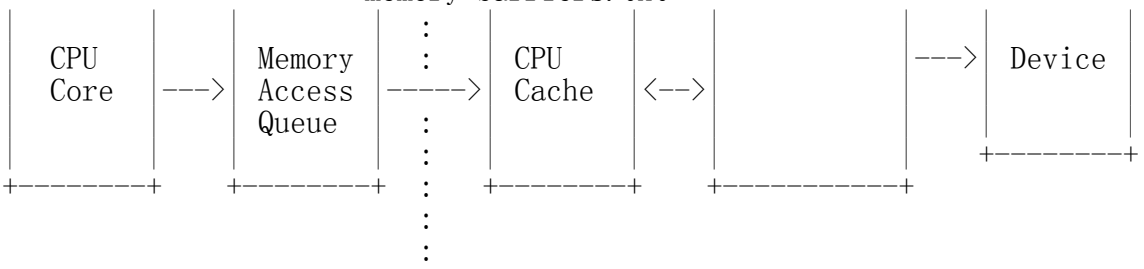
Similarly, it has to be assumed that compiler might reorder the instruction stream in any way it sees fit, again provided the appearance of causality is maintained.

============================
THE EFFECTS OF THE CPU CACHE
============================

The way cached memory operations are perceived across the system is affected to a certain extent by the caches that lie between CPUs and memory, and by the memory coherence system that maintains the consistency of state in the system.

As far as the way a CPU interacts with another part of the system through the caches goes, the memory system has to include the CPU's caches, and memory barriers for the most part act at the interface between the CPU and its cache (memory barriers logically act on the dotted line in the following diagram):





Although any particular load or store may not actually appear outside of the CPU that issued it since it may have been satisfied within the CPU's own cache, it will still appear as if the full memory access had taken place as far as the other CPUs are concerned since the cache coherency mechanisms will migrate the cacheline over to the accessing CPU and propagate the effects upon conflict.

The CPU core may execute instructions in any order it deems fit, provided the expected program causality appears to be maintained. Some of the instructions generate load and store operations which then go into the queue of memory accesses to be performed. The core may place these in the queue in any order it wishes, and continue execution until it is forced to wait for an instruction to complete.

What memory barriers are concerned with is controlling the order in which accesses cross from the CPU side of things to the memory side of things, and the order in which the effects are perceived to happen by the other observers in the system.

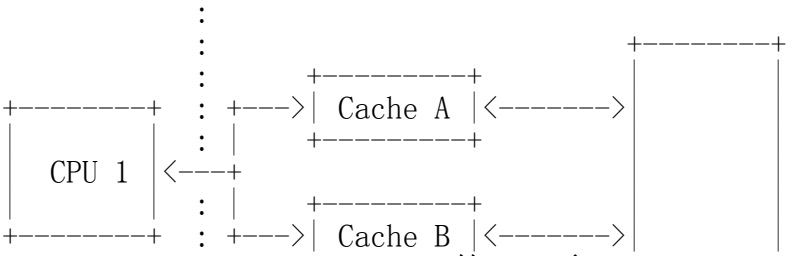
[!] Memory barriers are _not_ needed within a given CPU, as CPUs always see their own loads and stores as if they had happened in program order.

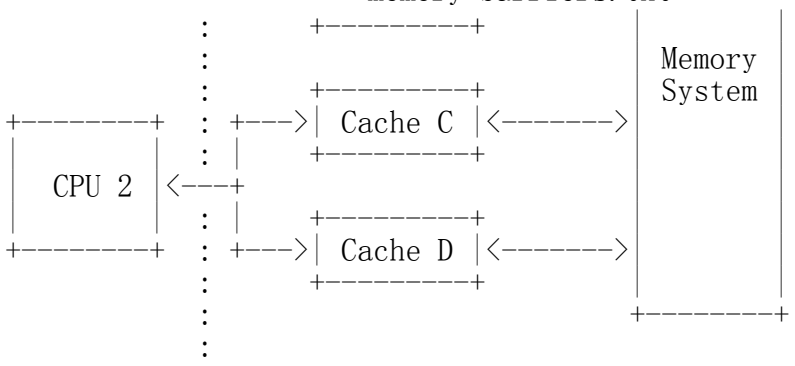
[!] MMIO or other device accesses may bypass the cache system. This depends on the properties of the memory window through which devices are accessed and/or the use of any special device communication instructions the CPU may have.

CACHE COHERENCY

Life isn't quite as simple as it may appear above, however: for while the caches are expected to be coherent, there's no guarantee that that coherency will be ordered. This means that whilst changes made on one CPU will eventually become visible on all CPUs, there's no guarantee that they will become apparent in the same order on those other CPUs.

Consider dealing with a system that has a pair of CPUs (1 & 2), each of which has a pair of parallel data caches (CPU 1 has A/B, and CPU 2 has C/D):





Imagine the system has the following properties:

(*) an odd-numbered cache line may be in cache A, cache C or it may still be resident in memory;

(*) an even-numbered cache line may be in cache B, cache D or it may still be resident in memory;

(*) whilst the CPU core is interrogating one cache, the other cache may be making use of the bus to access the rest of the system - perhaps to displace a dirty cacheline or to do a speculative load;

(*) each cache has a queue of operations that need to be applied to that cache to maintain coherency with the rest of the system;

(*) the coherency queue is not flushed by normal loads to lines already present in the cache, even though the contents of the queue may potentially affect those loads.

Imagine, then, that two writes are made on the first CPU, with a write barrier between them to guarantee that they will appear to reach that CPU's caches in the requisite order:

```
        CPU 1           CPU 2           COMMENT
        =============== =============== =======================================
                                        u == 0, v == 1 and p == &u, q == &u
        v = 2;
        smp_wmb();                      Make sure change to v is visible before
                                         change to p
        <A:modify v=2>                  v is now in cache A exclusively
        p = &v;
        <B:modify p=&v>                 p is now in cache B exclusively
```

The write memory barrier forces the other CPUs in the system to perceive that the local CPU's caches have apparently been updated in the correct order. But now imagine that the second CPU wants to read those values:

```
        CPU 1           CPU 2           COMMENT
        =============== =============== =======================================
        ...
                        q = p;
                        x = *q;
```

The above pair of reads may then fail to happen in the expected order, as the

cacheline holding p may get updated in one of the second CPU's caches whilst the update to the cacheline holding v is delayed in the other of the second CPU's caches by some other cache event:

```
        CPU 1           CPU 2           COMMENT
        =============== =============== =======================================
                                        u == 0, v == 1 and p == &u, q == &u
        v = 2;
        smp_wmb();
        <A:modify v=2>  <C:busy>
                        <C:queue v=2>
        p = &v;         q = p;
                        <D:request p>
        <B:modify p=&v> <D:commit p=&v>
                        <D:read p>
                        x = *q;
                        <C:read *q>     Reads from v before v updated in cache
                        <C:unbusy>
                        <C:commit v=2>
```

Basically, whilst both cachelines will be updated on CPU 2 eventually, there's no guarantee that, without intervention, the order of update will be the same as that committed on CPU 1.

To intervene, we need to interpolate a data dependency barrier or a read barrier between the loads. This will force the cache to commit its coherency queue before processing any further requests:

```
        CPU 1           CPU 2           COMMENT
        =============== =============== =======================================
                                        u == 0, v == 1 and p == &u, q == &u
        v = 2;
        smp_wmb();
        <A:modify v=2>  <C:busy>
                        <C:queue v=2>
        p = &v;         q = p;
                        <D:request p>
        <B:modify p=&v> <D:commit p=&v>
                        <D:read p>
                        smp_read_barrier_depends()
                        <C:unbusy>
                        <C:commit v=2>
                        x = *q;
                        <C:read *q>     Reads from v after v updated in cache
```

This sort of problem can be encountered on DEC Alpha processors as they have a split cache that improves performance by making better use of the data bus. Whilst most CPUs do imply a data dependency barrier on the read when a memory access depends on a read, not all do, so it may not be relied on.

Other CPUs may also have split caches, but must coordinate between the various cachelets for normal memory accesses. The semantics of the Alpha removes the need for coordination in the absence of memory barriers.

CACHE COHERENCY VS DMA

Not all systems maintain cache coherency with respect to devices doing DMA. In such cases, a device attempting DMA may obtain stale data from RAM because dirty cache lines may be resident in the caches of various CPUs, and may not have been written back to RAM yet. To deal with this, the appropriate part of the kernel must flush the overlapping bits of cache on each CPU (and maybe invalidate them as well).

In addition, the data DMA'd to RAM by a device may be overwritten by dirty cache lines being written back to RAM from a CPU's cache after the device has installed its own data, or cache lines present in the CPU's cache may simply obscure the fact that RAM has been updated, until at such time as the cacheline is discarded from the CPU's cache and reloaded. To deal with this, the appropriate part of the kernel must invalidate the overlapping bits of the cache on each CPU.

See Documentation/cachetlb.txt for more information on cache management.

CACHE COHERENCY VS MMIO

Memory mapped I/O usually takes place through memory locations that are part of a window in the CPU's memory space that has different properties assigned than the usual RAM directed window.

Amongst these properties is usually the fact that such accesses bypass the caching entirely and go directly to the device buses. This means MMIO accesses may, in effect, overtake accesses to cached memory that were emitted earlier. A memory barrier isn't sufficient in such a case, but rather the cache must be flushed between the cached memory write and the MMIO access if the two are in any way dependent.

==========================
THE THINGS CPUS GET UP TO
==========================

A programmer might take it for granted that the CPU will perform memory operations in exactly the order specified, so that if the CPU is, for example, given the following piece of code to execute:

```
a = *A;
*B = b;
c = *C;
d = *D;
*E = e;
```

they would then expect that the CPU will complete the memory operation for each instruction before moving on to the next one, leading to a definite sequence of operations as seen by external observers in the system:

```
LOAD *A, STORE *B, LOAD *C, LOAD *D, STORE *E.
```

Reality is, of course, much messier. With many CPUs and compilers, the above assumption doesn't hold because:

 (*) loads are more likely to need to be completed immediately to permit
     execution progress, whereas stores can often be deferred without a
     problem;

 (*) loads may be done speculatively, and the result discarded should it prove
     to have been unnecessary;

 (*) loads may be done speculatively, leading to the result having been fetched
     at the wrong time in the expected sequence of events;

 (*) the order of the memory accesses may be rearranged to promote better use
     of the CPU buses and caches;

 (*) loads and stores may be combined to improve performance when talking to
     memory or I/O hardware that can do batched accesses of adjacent locations,
     thus cutting down on transaction setup costs (memory and PCI devices may
     both be able to do this); and

 (*) the CPU's data cache may affect the ordering, and whilst cache-coherency
     mechanisms may alleviate this - once the store has actually hit the cache
     - there's no guarantee that the coherency management will be propagated in
     order to other CPUs.

So what another CPU, say, might actually observe from the above piece of code is:

```
LOAD *A, ..., LOAD {*C,*D}, STORE *E, STORE *B

(Where "LOAD {*C,*D}" is a combined load)
```

However, it is guaranteed that a CPU will be self-consistent: it will see its _own_ accesses appear to be correctly ordered, without the need for a memory barrier. For instance with the following code:

```
U = *A;
*A = V;
*A = W;
X = *A;
*A = Y;
Z = *A;
```

and assuming no intervention by an external influence, it can be assumed that the final result will appear to be:

```
U == the original value of *A
X == W
Z == Y
*A == Y
```

The code above may cause the CPU to generate the full sequence of memory

accesses:

```
U=LOAD *A, STORE *A=V, STORE *A=W, X=LOAD *A, STORE *A=Y, Z=LOAD *A
```

in that order, but, without intervention, the sequence may have almost any combination of elements combined or discarded, provided the program's view of the world remains consistent.

The compiler may also combine, discard or defer elements of the sequence before the CPU even sees them.

For instance:

```
*A = V;
*A = W;
```

may be reduced to:

```
*A = W;
```

since, without a write barrier, it can be assumed that the effect of the storage of V to *A is lost. Similarly:

```
*A = Y;
Z = *A;
```

may, without a memory barrier, be reduced to:

```
*A = Y;
Z = Y;
```

and the LOAD operation never appear outside of the CPU.

AND THEN THERE'S THE ALPHA

The DEC Alpha CPU is one of the most relaxed CPUs there is. Not only that, some versions of the Alpha CPU have a split data cache, permitting them to have two semantically-related cache lines updated at separate times. This is where the data dependency barrier really becomes necessary as this synchronises both caches with the memory coherence system, thus making it seem like pointer changes vs new data occur in the right order.

The Alpha defines the Linux kernel's memory barrier model.

See the subsection on "Cache Coherency" above.

EXAMPLE USES

CIRCULAR BUFFERS

Memory barriers can be used to implement circular buffering without the need of a lock to serialise the producer with the consumer. See:

        Documentation/circular-buffers.txt

for details.

==========
REFERENCES
==========

Alpha AXP Architecture Reference Manual, Second Edition (Sites & Witek, Digital Press)
        Chapter 5.2: Physical Address Space Characteristics
        Chapter 5.4: Caches and Write Buffers
        Chapter 5.5: Data Sharing
        Chapter 5.6: Read/Write Ordering

AMD64 Architecture Programmer's Manual Volume 2: System Programming
        Chapter 7.1: Memory-Access Ordering
        Chapter 7.4: Buffering and Combining Memory Writes

IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide
        Chapter 7.1: Locked Atomic Operations
        Chapter 7.2: Memory Ordering
        Chapter 7.4: Serializing Instructions

The SPARC Architecture Manual, Version 9
        Chapter 8: Memory Models
        Appendix D: Formal Specification of the Memory Models
        Appendix J: Programming with the Memory Models

UltraSPARC Programmer Reference Manual
        Chapter 5: Memory Accesses and Cacheability
        Chapter 15: Sparc-V9 Memory Models

UltraSPARC III Cu User's Manual
        Chapter 9: Memory Models

UltraSPARC IIIi Processor User's Manual
        Chapter 8: Memory Models

UltraSPARC Architecture 2005
        Chapter 9: Memory
        Appendix D: Formal Specifications of the Memory Models

UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005
        Chapter 8: Memory Models
        Appendix F: Caches and Cache Coherency

Solaris Internals, Core Kernel Architecture, p63-68:
        Chapter 3.3: Hardware Considerations for Locks and
                        Synchronization

Unix Systems for Modern Architectures, Symmetric Multiprocessing and Caching
for Kernel Programmers:
        Chapter 13: Other Memory Models

Intel Itanium Architecture Software Developer's Manual: Volume 1:
        Section 2.6: Speculation
        Section 4.4: Memory Access