Memory Barriers in the Linux Kernel
Semantics and Practices

Embedded Linux Conference – April 2016. San Diego, CA.

Davidlohr Bueso <dave@stgolabs.net>
SUSE Labs.
Agenda

1. Introduction
   • Reordering Examples
   • Underlying need for memory barriers

2. Barriers in the kernel
   • Building blocks
   • Implicit barriers
   • Atomic operations
   • Acquire/release semantics.
References

i. David Howells, Paul E. McKenney. Linux Kernel source: Documentation/memory-barriers.txt


Flagship Example

\[ A = 0, \quad B = 0 \] (shared variables)

CPU0
\[
\begin{align*}
A &= 1 \\
x &= B
\end{align*}
\]

CPU1
\[
\begin{align*}
B &= 1 \\
y &= A
\end{align*}
\]
Flagship Example

A = 0, B = 0 (shared variables)

CPU0          CPU1
A = 1          B = 1
x = B          y = A

(x, y) =
Flagship Example

\[ A = 0, \ B = 0 \] \text{(shared variables)}

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A = 1 )</td>
<td>( B = 1 )</td>
</tr>
<tr>
<td>( x = B )</td>
<td>( y = A )</td>
</tr>
</tbody>
</table>

\[ (x, \ y) = (0, 1) \]

\[ A = 1 \]
\[ x = B \]
\[ B = 1 \]
\[ y = A \]
Flagship Example

\[ A = 0, \ B = 0 \ (\text{shared variables}) \]

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1</td>
<td>B = 1</td>
</tr>
<tr>
<td>x = B</td>
<td>y = A</td>
</tr>
<tr>
<td></td>
<td>(x, y) = (1, 0)</td>
</tr>
</tbody>
</table>

\[ B = 1 \\
\ A = 1 \\
\ x = B \]

\[ (0, 1) \]
Flagship Example

A = 0, B = 0 (shared variables)

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1</td>
<td>B = 1</td>
</tr>
<tr>
<td>x = B</td>
<td>y = A</td>
</tr>
</tbody>
</table>

A = 1

B = 1

y = A

x = B
## Flagship Example

\[ A = 0, \ B = 0 \] (shared variables)

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
<th>( (0, 1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A = 1 )</td>
<td>( \ B = 1 )</td>
<td>( (x, \ y) = (1, 0) )</td>
</tr>
<tr>
<td>( x = B )</td>
<td>( y = A )</td>
<td>( (1, 1) )</td>
</tr>
<tr>
<td>( x = B )</td>
<td>( y = A )</td>
<td>( (0, 0) )</td>
</tr>
<tr>
<td>( A = 1 )</td>
<td>( \ B = 1 )</td>
<td></td>
</tr>
</tbody>
</table>
Memory Consistency Models

- Most modern multicore systems are *coherent* but not *consistent*.
  - Same address is subject to the cache coherency protocol.
- Describes what the CPU can do regarding instruction ordering across addresses.
  - Helps programmers make sense of the world.
  - CPU is not aware if application is single or multi-threaded. When optimizing, it only ensures single threaded correctness.
Sequential Consistency (SC)

“A multiprocessor is sequentially consistent if the result of any execution is the same as some sequential order, and within any processor, the operations are executed in program order”

• Intuitively a programmer's ideal scenario.
  - The instructions are executed by the same CPU in the order in which it was written.
  - All processes see the same interleaving of operations.
Total Store Order (TSO)

- SPARC, x86 (Intel, AMD)
- Similar to SC, but:
  - Loads may be reordered with writes.

[1] A
[1] B
[s] B
[1] B
[1] B
[s] C
[1] B
[1] B
[s] A
[s] B
Total Store Order (TSO)

- SPARC, x86 (Intel, AMD)
- Similar to SC, but:
  - Loads may be reordered with writes.

```
[1] A
[1] B
[s] B
[1] B
[s] C
[1] B
[s] A
[s] B
L→L
```
Total Store Order (TSO)

- SPARC, x86 (Intel, AMD)
- Similar to SC, but:
  - Loads may be reordered with writes.

\[
\begin{align*}
\text{[1]} & \quad \text{A} \\
\text{[1]} & \quad \text{B} \\
\text{[s]} & \quad \text{B} \\
\text{[1]} & \quad \text{B} \\
\text{[1]} & \quad \text{B} \\
\text{[s]} & \quad \text{C} \\
\text{[1]} & \quad \text{B} \\
\text{[s]} & \quad \text{A} \\
\text{[s]} & \quad \text{B}
\end{align*}
\]

\[
\begin{align*}
L \rightarrow L \\
S \rightarrow S
\end{align*}
\]
Total Store Order (TSO)

• SPARC, x86 (Intel, AMD)

• Similar to SC, but:
  - Loads may be reordered with writes.

\[
\begin{align*}
[1] & \quad A \\
[1] & \quad B \\
[s] & \quad B \\
[1] & \quad B \\
[s] & \quad C \\
[1] & \quad B \\
[s] & \quad A \\
[s] & \quad B \\
\end{align*}
\]

\[
\begin{align*}
L & \rightarrow L \\
L & \rightarrow S \\
S & \rightarrow S
\end{align*}
\]
Total Store Order (TSO)

- SPARC, x86 (Intel, AMD)
- Similar to SC, but:
  - Loads may be reordered with writes.

\[
\begin{align*}
[1] & \quad A & \quad L \rightarrow L \\
[1] & \quad B & \\
[s] & \quad B & \\
[1] & \quad B & \quad L \rightarrow S \\
[s] & \quad C & \quad S \rightarrow L \\
[1] & \quad B & \\
[s] & \quad A & \quad S \rightarrow S \\
[s] & \quad B &
\end{align*}
\]
Relaxed Models

• Arbitrary reorder limited only by explicit memory-barrier instructions.

• ARM, Power, tilera, Alpha.
Fixing the Example

A = 0, B = 0 (shared variables)

CPU0
A = 1
x = B

CPU1
B = 1
y = A
Fixing the Example

A = 0, B = 0 (shared variables)

CPU0
A = 1
<MB>
x = B

CPU1
B = 1
<MB>
y = A
## Fixing the Example

\[ A = 0, \ B = 0 \ (\text{shared variables}) \]

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A = 1 )</td>
<td>( B = 1 )</td>
</tr>
<tr>
<td>( &lt;\text{MB}&gt; )</td>
<td>( &lt;\text{MB}&gt; )</td>
</tr>
<tr>
<td>( x = B )</td>
<td>( y = A )</td>
</tr>
</tbody>
</table>

- Compiler barrier
Fixing the Example

A = 0, B = 0 (shared variables)

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1</td>
<td>B = 1</td>
</tr>
<tr>
<td>&lt;MB&gt;</td>
<td>&lt;MB&gt;</td>
</tr>
<tr>
<td>x = B</td>
<td>y = A</td>
</tr>
</tbody>
</table>

- Compiler barrier
- Mandatory barriers (general+rw)
## Fixing the Example

A = 0, B = 0 (shared variables)

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1</td>
<td>B = 1</td>
</tr>
<tr>
<td>&lt;MB&gt;</td>
<td>&lt;MB&gt;</td>
</tr>
<tr>
<td>x = B</td>
<td>y = A</td>
</tr>
</tbody>
</table>

- Compiler barrier
- Mandatory barriers (general+rw)
- SMP-conditional barriers
Fixing the Example

A = 0, B = 0 (shared variables)

CPU0                    CPU1
A = 1                  B = 1
<MB>                  <MB>
x = B                 y = A

• Compiler barrier
• Mandatory barriers (general+rw)
• SMP-conditional barriers
• acquire/release
Fixing the Example

A = 0, B = 0 (shared variables)

CPU0
A = 1
<MB>
x = B

CPU1
B = 1
<MB>
y = A

- Compiler barrier
- Mandatory barriers (general+rw)
- SMP-conditional barriers
- acquire/release
- Data dependency barriers
- Device barriers
Barriers in the Linux Kernel
Abstracting Architectures

• Most kernel programmers need not worry about ordering specifics of every architecture.
  - Some notion of barrier usage is handy nonetheless – implicit vs explicit, semantics, etc.

• Linux must handle the CPU's memory ordering specifics in a portable way with LCD semantics of memory barriers.
  - CPU appears to execute in program order.
  - Single variable consistency.
  - Barriers operate in pairs.
  - Sufficient to implement synchronization primitives.
Abstracting Architectures

- Each architecture must implement its own calls or otherwise default to the generic and highly unoptimized behavior.

- `<arch/xxx/include/asm/barriers.h>` will always define the low-level CPU specifics, then rely on `<include/asm-generic/barriers.h>`
A Note on barrier()

- Prevents the compiler from getting *smart*, acting as a general barrier.

- Within a loop forces the compiler to reload conditional variables – **READ/WRITE_ONCE**.
Implicit Barriers

• Calls that have implied barriers, the caller can safely rely on:
  - Locking functions
  - Scheduler functions
  - Interrupt disabling functions
  - Others.
Sleeping/Waking

- Extremely common task in the kernel and flagship example of flag-based CPU-CPU interaction.

```c
CPU0
while (!done) {
    schedule();
    current->state = ...;
}
CPU1
done = true;
wake_up_process(t);
```
Sleeping/Waking

- Extremely common task in the kernel and flagship example of flag-based CPU-CPU interaction.

```c
CPU0

while (!done) {
    done = true;
    schedule();
    current->state = ...
    set_current_state(...);
}

CPU1

wake_up_process(t);
```
Sleeping/Waking

- Extremely common task in the kernel and flagship example of flag-based CPU-CPU interaction.

```c
CPU0

while (!done) {
    schedule();
    current->state = ...;
    set_current_state(...);
}

CPU1

done = true;
wake_up_process(t);
smp_store_mb():
    [s] → state = ...
smp_mb()
```
Atomic Operations

• Any atomic operation that modifies some state in memory and returns information about the state can potentially imply a SMP barrier:
  - smp_mb() on each side of the actual operation
    
    [atomic_*__]xchg()
    atomic_*__return()
    atomic_*__and_test()
    atomic_*__add_negative()
Atomic Operations

- Any atomic operation that modifies some state in memory and returns information about the state can potentially imply a SMP barrier:
  - `smp_mb()` on each side of the actual operation
    ```c
    [atomic_*_]xchg()
    atomic_*_return()
    atomic_*_and_test()
    atomic_*_add_negative()
    ```
  - Conditional calls imply barriers only when successful.
    ```c
    [atomic_*_]cmpxchg()
    atomic_*_add_unless()
    ```
Atomic Operations

• Most basic of operations therefore do not imply barriers.

• Many contexts can require barriers:

```c
cpumask_set_cpu(cpu, vec->mask);
/*
 * When adding a new vector, we update the mask first,
 * do a write memory barrier, and then update the count, to
 * make sure the vector is visible when count is set.
 */
smp_mb__before_atomic();
atomic_inc(&(vec)->count);
```
Atomic Operations

• Most basic of operations therefore do not imply barriers.

• Many contexts can require barriers:

```c
/*
 * When removing from the vector, we decrement the counter first
 * do a memory barrier and then clear the mask.
 */
atomic_dec(&(vec)->count);
smp_mb__after_atomic();
cpumask_clear_cpu(cpu, vec->mask);
```
Acquire/Release Semantics

- One way barriers.
- Passing information reliably between threads about a variable.
  - Ideal in producer/consumer type situations (pairing!!).
  - After an ACQUIRE on a given variable, all memory accesses preceding any prior RELEASE on that same variable are guaranteed to be visible.
  - All accesses of all previous critical sections for that variable are guaranteed to have completed.
  - C++11's `memory_order_acquire, memory_order_release` and `memory_order_relaxed`.
Acquire/Release Semantics

CPU0

```
spin_lock(&l)
CR
spin_unlock(&l)
```

CPU1

```
spin_lock(&l)
CR
spin_unlock(&l)
```
Acquire/Release Semantics

CPU0

spin_lock(&l)

spin_unlock(&l)

CPU1

spin_lock(&l)

spin_unlock(&l)

\[ \text{smp\_store\_release}(\text{lock} \rightarrow \text{val}, 0) \leftrightarrow \text{cmpxchg\_acquire}(\text{lock} \rightarrow \text{val}, 0, \text{LOCKED}) \]
Acquire/Release Semantics

CPU0

spin_lock(&l)

CR

RELEASE

spin_unlock(&l)

CPU1

spin_lock(&l)

CR

ACQUIRE

spin_unlock(&l)

\[
\text{smp\_store\_release}(\text{lock}\rightarrow\text{val}, 0) \leftrightarrow \text{cmpxchg\_acquire}(\text{lock}\rightarrow\text{val}, 0, \text{LOCKED})
\]
Acquire/Release Semantics

CPU0

\[
\text{spin\_lock}(&l) \quad \text{CR} \quad \text{spin\_unlock}(&l) \quad \text{CR}
\]

CPU1

\[
\text{spin\_lock}(&l) \quad \text{spin\_lock}(&l) \quad \text{spin\_unlock}(&l) \quad \text{spin\_unlock}(&l)
\]

\[
\text{RELEASE} \ (\text{LS}, \text{SS}) \quad \text{ACQUIRE} \ (\text{LL}, \text{LS})
\]

\[
\text{smp\_store\_release}(\text{lock}\rightarrow\text{val}, \ 0) \leftrightarrow \text{cmpxchg\_acquire}(\text{lock}\rightarrow\text{val}, \ 0, \ \text{LOCKED})
\]
Acquire/Release Semantics

CPU0

spin_lock(&l)

CPU1

spin_lock(&l)

spin_unlock(&l)

spin_unlock(&l)

smp_store_release(lock→val, 0) <-> cmpxchg_acquire(lock→val, 0, LOCKED)
Acquire/Release Semantics

\[\text{smp\_store\_release}(\text{lock}\rightarrow\text{val}, 0) \leftrightarrow \text{cmpxchg\_acquire}(\text{lock}\rightarrow\text{val}, 0, \text{LOCKED})\]
Acquire/Release Semantics

smp_store_release(lock->val, 0) <-> cmpxchg_acquire(lock->val, 0, LOCKED)
Acquire/Release Semantics

• Regular atomic/RMW calls have been fine grained for archs that support strict acquire/release semantics.
  
  \begin{align*}
  \text{cmpxchg}() & \quad \text{smp\_load\_acquire}() \\
  \text{cmpxchg\_acquire}() & \quad \text{smp\_cond\_acquire}() \\
  \text{cmpxchg\_release}() & \quad \text{smp\_store\_release}() \\
  \text{cmpxchg\_relaxed}() & \quad \text{smp\_load\_acquire}()
  \end{align*}

• Currently only used by arm64 and PPC.
  - LDAR/STLR
Acquire/Release Semantics

• These are **minimal** guarantees.
  - Ensuring barriers on both sides of a lock operation will require therefore, full barrier semantics:
    
    ```
    smp_mb__before_spinlock()
    smp_mb__after_spinlock()
    ```

• Certainly not limited to locking.
  - perf, IPI paths, scheduler, tty, etc.
Acquire/Release Semantics

- Busy-waiting on a variable that requires ACQUIRE semantics:

  CPU0
  
  while (!done)
      cpu_relax();
  
  smp_rmb();

  CPU1
  
  smp_store_release(done, 1);
Acquire/Release Semantics

- Busy-waiting on a variable that requires ACQUIRE semantics:

  CPU0

  while (!done)
    cpu_relax();
    smp_rmb();
  [LS]

  CPU1

  smp_store_release(done, 1);
  [LL]
Acquire/Release Semantics

• Busy-waiting on a variable that requires ACQUIRE semantics:

CPU0

while (!done)
    cpu_relax();
    smp_rmb();

CPU1

    [LS] smp_store_release(done, 1);
    [LL]

    smp_load_acquire(!done);
Acquire/Release Semantics

• Busy-waiting on a variable that requires ACQUIRE semantics:

```c
CPU0
while (!done)
    cpu_relax();
    [LS]  smp_store_release(done, 1);
CPU1
smp_rmb();  [LL]
```

• Fine-graining SMP barriers while a performance optimization, makes it harder for kernel programmers.
Concluding Remarks

• Assume nothing.

• Read memory-barriers.txt

• Use barrier pairings.

• Comment barriers.
Thank you.