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

ii. Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?*

iii. Paul E. McKenney. Memory Barriers: a Hardware View for Software Hackers. June 2010.

iv. Sorin, Hill, Wood. *A Primer on Memory Consistency and Cache Coherence.* Synthesis Lectures on Computer Architecture. 2011.

Flagship Example

```
A = 0, B = 0 (shared variables)

CPU0            CPU1

A = 1           B = 1
x = B           y = A
```

Flagship Example

```
A = 0, B = 0 (shared variables)

CPU0            CPU1

A = 1           B = 1           (x, y) =
x = B           y = A
```

Flagship Example

```
A = 0, B = 0 (shared variables)

CPU0            CPU1                        (0, 1)

A = 1           B = 1           (x, y) =
x = B           y = A
```

```
A = 1
x = B
                B = 1
                y = A
```

Flagship Example

```
A = 0, B = 0 (shared variables)

CPU0            CPU1                        (0, 1)

A = 1           B = 1          (x, y) =     (1, 0)
x = B           y = A
```

```
                B = 1
                y = A
A = 1
x = B
```

Flagship Example

```
A = 0, B = 0 (shared variables)

CPU0            CPU1                        (0, 1)

A = 1           B = 1          (x, y) =     (1, 0)
x = B           y = A
                                            (1, 1)
```

```
A = 1
                B = 1
                y = A
x = B
```

Flagship Example

A = 0, B = 0 (shared variables)

CPU0	CPU1	(x, y) =
A = 1	B = 1	(0, 1)
x = B	y = A	(1, 0)
		(1, 1)
		(0, 0)

x = B
A = 1

y = A
B = 1

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*"
>
> – Lamport, 1979.

- 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.

```
[l] A
[l] B
[s] B
[l] B
[s] C
[l] B
[s] A
[s] B
```

Total Store Order (TSO)

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

```
[l] A
[l] B
```

L→L

```
[s] B
[l] B
[s] C
[l] B
[s] A
[s] B
```

Total Store Order (TSO)

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

```
[l] A
[l] B
[s] B
[l] B
[s] C
[l] B
[s] A
[s] B
```

L→L

S→S

Total Store Order (TSO)

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

```
[l] A
[l] B      L→L

[s] B

[l] B      L→S
[s] C

[l] B

[s] A      S→S
[s] B
```

Total Store Order (TSO)

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

```
[l] A
[l] B        L→L

[s] B

[l] B        L→S
[s] C

[l] B        S→L

[s] A
[s] B        S→S
```

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            CPU1

A = 1           B = 1

x = B           y = A
```

Fixing the Example

```
A = 0, B = 0 (shared variables)

CPU0            CPU1

A = 1           B = 1
<MB>            <MB>
x = B           y = A
```

Fixing the Example

```
A = 0, B = 0 (shared variables)

CPU0            CPU1

A = 1           B = 1
<MB>            <MB>
x = B           y = A
```

- Compiler barrier

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)

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

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            CPU1

A = 1           B = 1
<MB>            <MB>
x = B           y = A
```

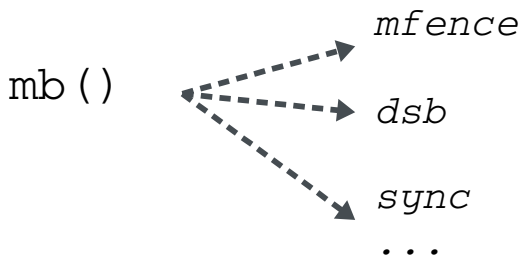
- Compiler barrier
- Mandatory barriers (general+rw)
- SMP-conditional barriers
- *acquire/release*
- Data dependency barriers
- *Device barriers*

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.

```
CPU0                                CPU1

while (!done) {                     done = true;
    schedule();                     wake_up_process(t);
    current→state = …;
}
```

Sleeping/Waking

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

```
CPU0                                CPU1

while (!done) {                     done = true;
    schedule();                     wake_up_process(t);
    current->state = …;
    set_current_state(…);
}
```

Sleeping/Waking

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

```
CPU0                                CPU1

while (!done) {                     done = true;
    schedule();                     wake_up_process(t);
    current→state = …;
    set_current_state(…);
}
```

```
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

```
[atomic_*_]xchg()
atomic_*_return()
atomic_*_and_test()
atomic_*_add_negative()
```

  - Conditional calls imply barriers only when successful.

```
[atomic_*_]cmpxchg()
atomic_*_add_unless()
```

Atomic Operations

- Most basic of operations therefore do not imply barriers.
- Many contexts can require barriers:

```
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

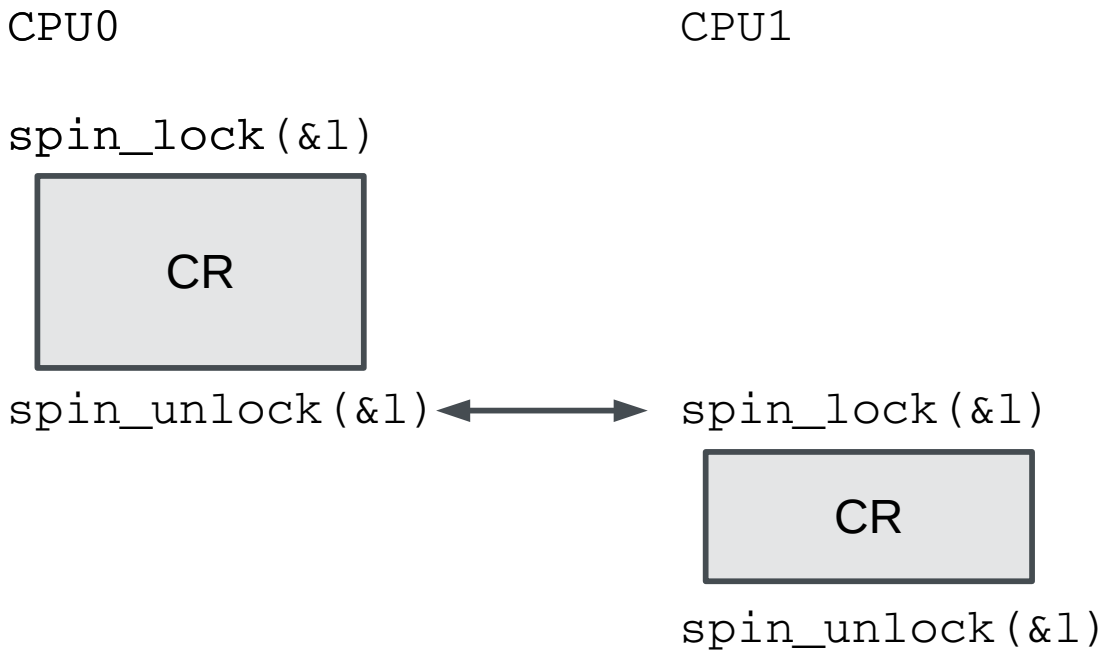
- Most basic of operations therefore do not imply barriers.
- Many contexts can require barriers:

```
/*
 * 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);
```

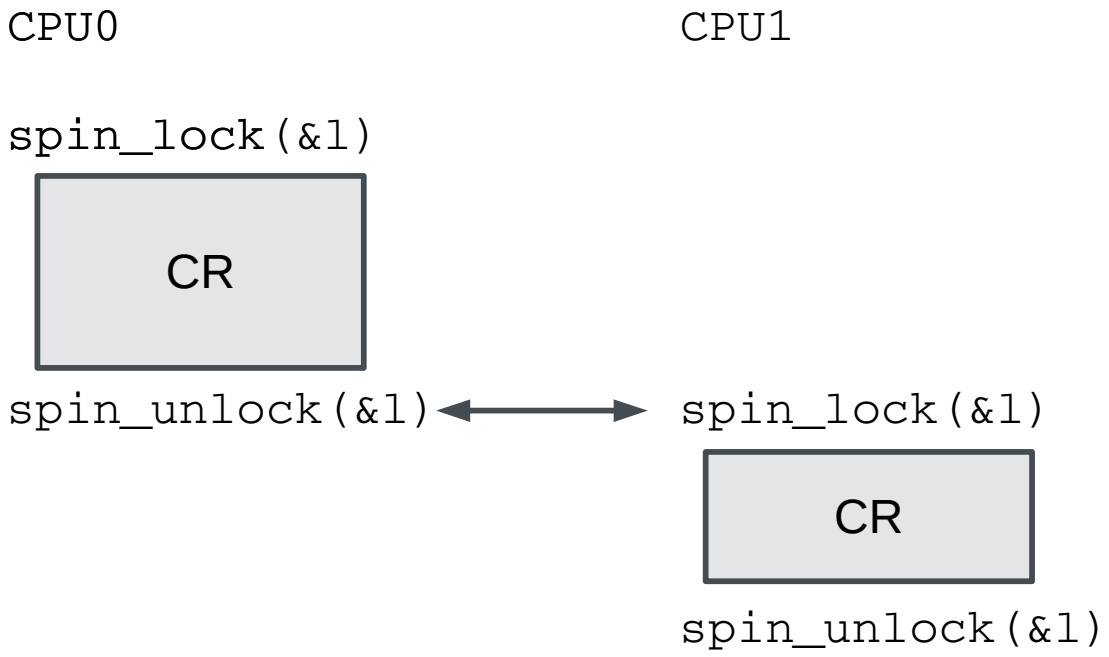
Acquire/Release Semantics

- 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

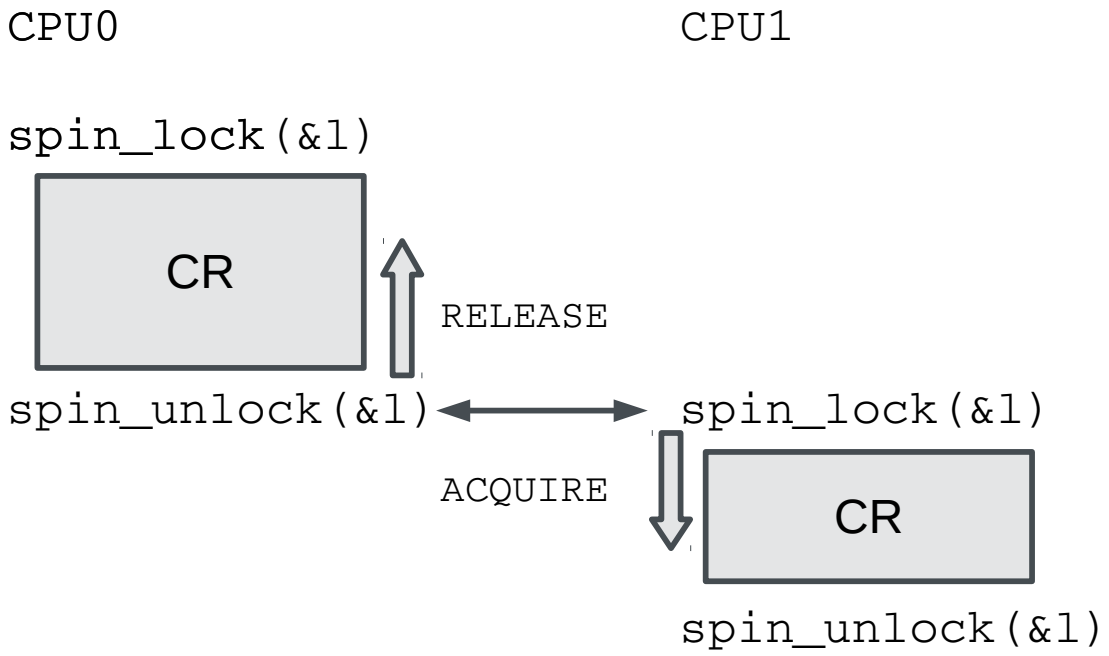


Acquire/Release Semantics



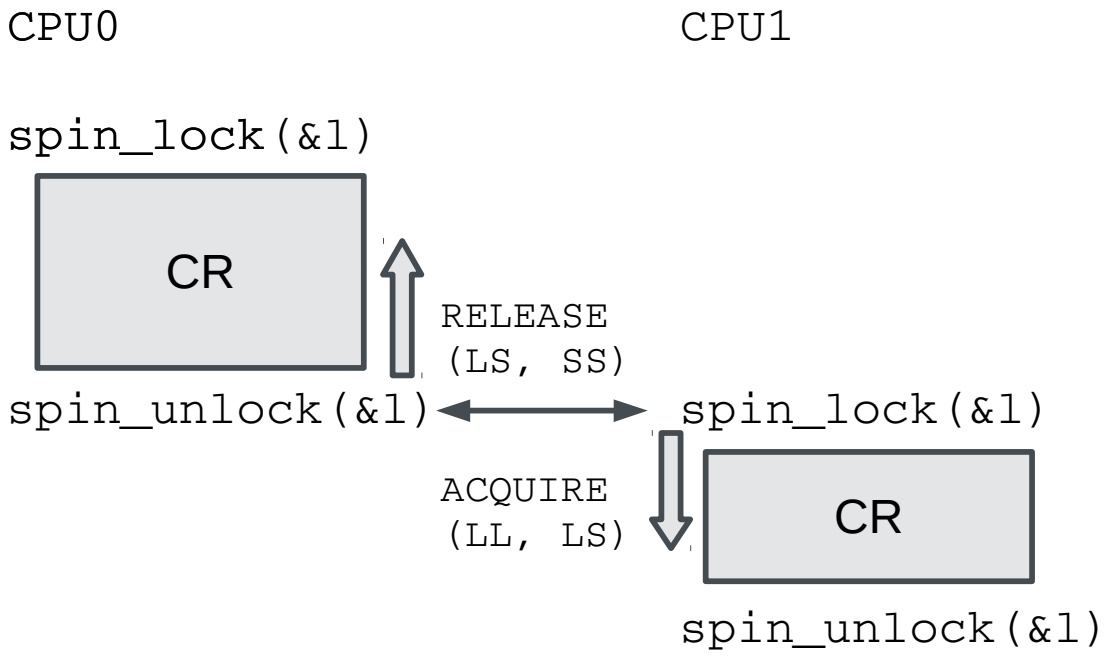
smp_store_release(lock→val, 0) <-> **cmpxchg_acquire**(lock→val, 0, LOCKED)

Acquire/Release Semantics



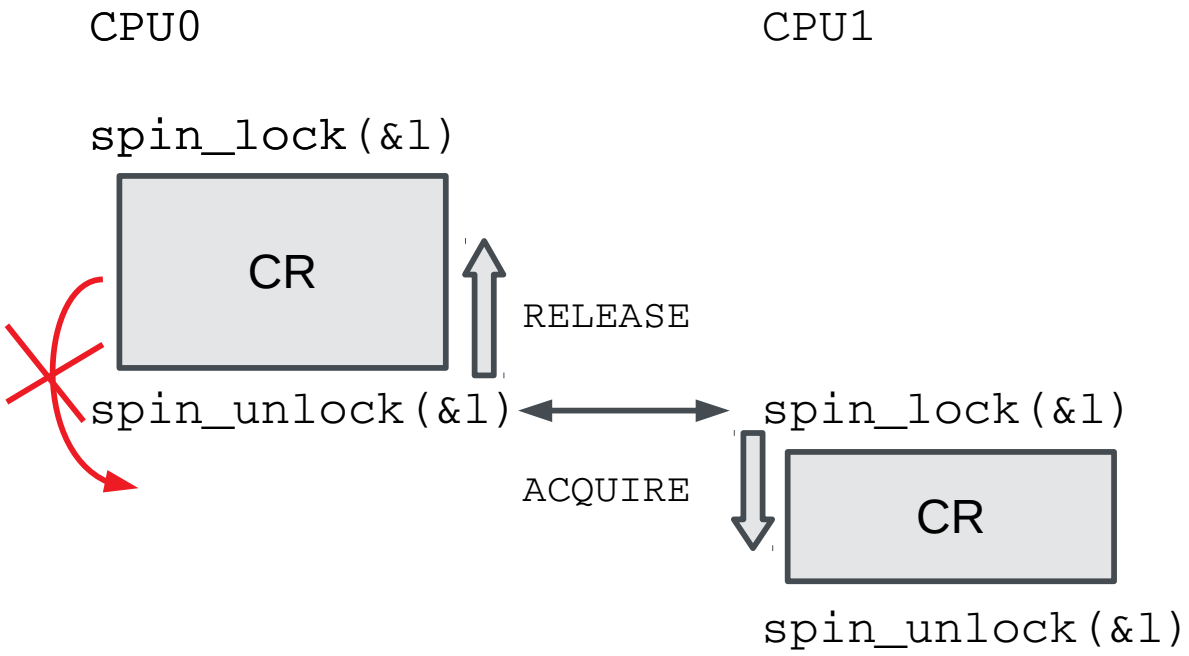
smp_store_release(lock→val, 0) <-> **cmpxchg_acquire**(lock→val, 0, LOCKED)

Acquire/Release Semantics



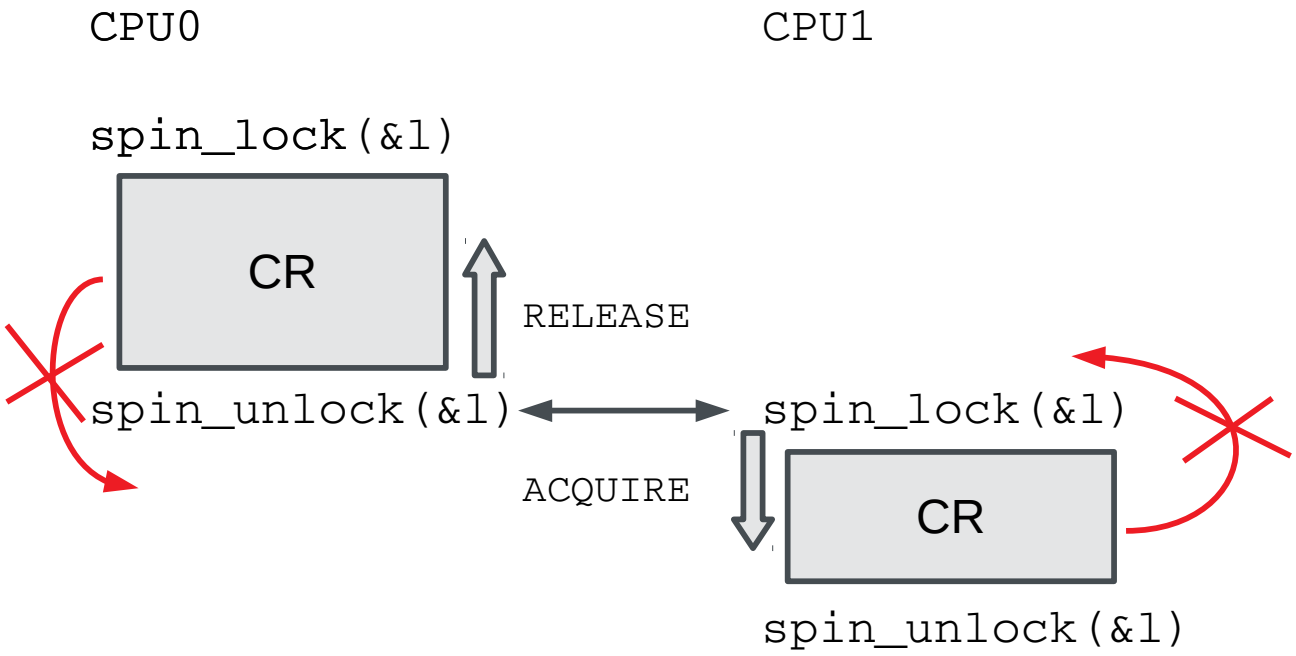
smp_store_release(lock→val, 0) <-> **cmpxchg_acquire**(lock→val, 0, LOCKED)

Acquire/Release Semantics



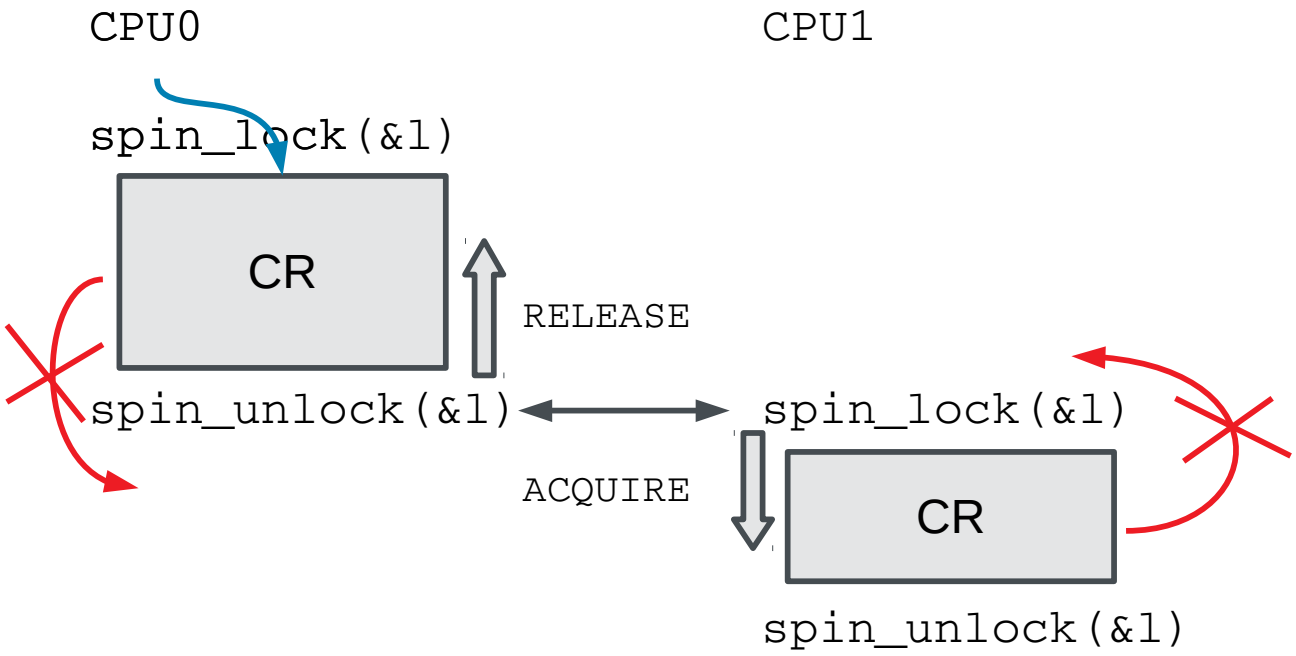
smp_store_release(lock→val, 0) <-> **cmpxchg_acquire**(lock→val, 0, LOCKED)

Acquire/Release Semantics



smp_store_release(lock→val, 0) <-> **cmpxchg_acquire**(lock→val, 0, 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.

```
cmpxchg()                    smp_load_acquire()
cmpxchg_acquire()            smp_cond_acquire()
cmpxchg_release()            smp_store_release()
cmpxchg_relaxed()
```

- 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                              CPU1

while (!done)
    cpu_relax();
                                  smp_store_release(done, 1);
smp_rmb();
```

Acquire/Release Semantics

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

```
CPU0                        CPU1

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

Acquire/Release Semantics

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

```
CPU0                        CPU1

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

```
smp_load_acquire(!done);
```

Acquire/Release Semantics

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

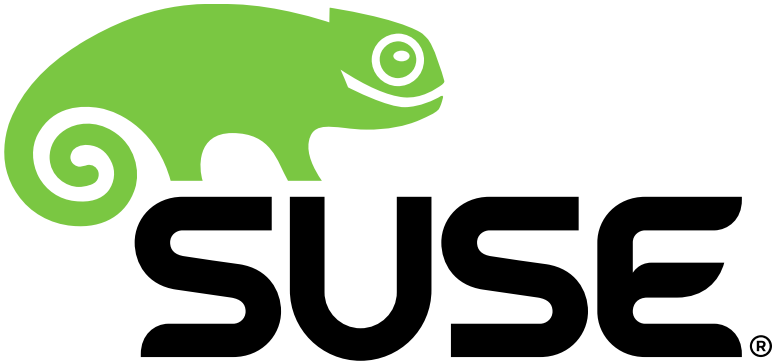
```
CPU0                        CPU1

while (!done)
    cpu_relax();
                    [LS]    smp_store_release(done, 1);
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.


SUSE®
We adapt. You succeed.