Concurrency with `tools/memory-model`

Andrea Parri

andrea.parri@amarulasolutions.com

Kernel Summit 2018

amarula solutions

. . . (part of) the LKMM subsystem

- Merged in 4.17
- $\sim$ 5000 LoC and documentation
- 10 maintainers, 2 reviewers

Motivations

Test it, stupid!

```
static int data = 0;
static int flag = 0;

void producer(void)
{
   WRITE_ONCE(data, 1);
   smp_store_release(&flag, 1);
}
```

```
void consumer(void)
{
   int r_flag;

   r_flag = smp_load_acquire(&flag);
   if (r_flag) {
      int r_data;

      r_data = READ_ONCE(data);
      WARN_ON(r_data == 0);
      /* process r_data */
   }
}
```

amarula solutions

Read the fine manual!

```
#define __smp_store_release(p, v)                                       \
do {                                                                    \
   union { typeof(*p) __val; char __c[1]; } __u =                       \
      { .__val = (__force typeof(*p)) (v) };                            \
   compiletime_assert_atomic_type(*p);                                  \
   switch (sizeof(*p)) {                                                \
   [...]                                                                \
   case 4:                                                              \
      asm volatile ("stlr %w1, %0"                                      \
                    : "=Q" (*p)                                         \
                    : "r" (*(__u32 *)__u.__c)                           \
                    : "memory");                                        \
      break;                                                            \
   [...]                                                                \
   }                                                                    \
} while (0)
```

amarula solutions

Are you for real?? (and gut feelings. . .)

- Joe Random Developer is likely *not going* to review 10+ implementations of `smp_store_release()`
- Architectures' maintainers are likely *not going* to review Joe's patches about his cool new feature `X`

The LKMM as an intermediary

The purpose of this document is twofold:

(1) to specify the minimum functionality that one can rely on for any particular barrier, and

(2) to provide a guide as to how to use the barriers that are available.

(from Documentation/memory-barriers.txt)

Basic usage

Litmus tests, aka querying the memory model

```
C producer-consumer

{
    int data = 0;
    int flag = 0;
}

P0(int *data, int *flag)
{
    WRITE_ONCE(*data, 1);
    smp_store_release(flag, 1);
}
```

```
P1(int *data, int *flag)
{
    int r_flag;
    int r_data = -1;

    r_flag = smp_load_acquire(flag);
    if (r_flag) {
        r_data = READ_ONCE(*data);
    }
}

exists (1:r_flag=1 /\ 1:r_data=0)
```

amarula solutions

Basic usage: reachable states

```
$ herd7 -conf linux-kernel.cfg producer-consumer.litmus
Test producer-consumer Allowed
States 2
1:r_data=-1; 1:r_flag=0;
1:r_data=1; 1:r_flag=1;
No
Witnesses
Positive: 0 Negative: 2
Condition exists (1:r_flag=1 /\ 1:r_data=0)
[...]
```

amarula solutions

The LKMM as a formal specification

```
This memory model can (roughly speaking) be thought of
as an automated version of memory-barriers.txt.  It is
written in the "cat" language, which is executable by
the externally provided "herd7" simulator [...]

                                                Paul E. McKenney
```

```
LIMITATIONS
===========
1. [...] but there is [...] code that uses bare C memory accesses
   [...]  this [...] in turn limits LKMM's ability to accurately
   model address, control, and data dependencies.
2. Multiple access sizes for a single variable are not supported
   and neither are misaligned or partially overlapping accesses.
3. Exceptions and interrupts are not modeled. [...]
4. I/O such as MMIO or DMA is not supported.
5. Self-modifying code [...] is not supported.
6. Complete modeling of all variants of atomic read-modify-write
   operations, locking primitives, and RCU is not provided. [...]
```

(from tools/memory-model/README)

Examples

Coherence

```
C read-read-coherence

{
    int x = 0;
}

P0(int *x)
{
    WRITE_ONCE(*x, 1);
}
```

```
P1(int *x)
{
    int r0;
    int r1;

    r0 = READ_ONCE(*x);
    r1 = READ_ONCE(*x);
}

exists (1:r0=1 /\ 1:r1=0)
```

This ‘exists’ clause can NOT be satisfied!

Execution and propagation (release and acquire)

```
C message-passing

{
    int x = 0;
    int y = 0;
}

P0(int *x, int *y)
{
    WRITE_ONCE(*x, 1);
    smp_store_release(y, 1);
}
```

```
P1(int *x, int *y)
{
    int r0;
    int r1;

    r0 = smp_load_acquire(y);
    r1 = READ_ONCE(*x);
}

exists (1:r0=1 /\ 1:r1=0)
```

This ‘exists’ clause can NOT be satisfied!

amarula solutions

Cumulativity

```
C release-is-(A-)cumulative

{
    int x = 0;
    int y = 0;
}

P0(int *x)
{
    WRITE_ONCE(*x, 1);
}

P1(int *x, int *y)
{
    int r0;

    r0 = READ_ONCE(*x);
    smp_store_release(y, 1);
}
```

```
P2(int *x, int *y)
{
    int r0;
    int r1;

    r0 = smp_load_acquire(y);
    r1 = READ_ONCE(*x);
}

exists (1:r0=1 /\ 2:r0=1 /\ 2:r1=0)
```

This ‘exists’ clause can NOT be satisfied!

amarula solutions

Execution and propagation (full memory barriers)

```
C store-buffering

{
    int x = 0;
    int y = 0;
}

P0(int *x, int *y)
{
    int r0;

    WRITE_ONCE(*x, 1);
    smp_mb();
    r0 = READ_ONCE(*y);
}
```

```
P1(int *x, int *y)
{
    int r0;

    WRITE_ONCE(*y, 1);
    smp_mb();
    r0 = READ_ONCE(*x);
}

exists (0:r0=0 /\ 1:r0=0)
```

This ‘exists’ clause can NOT be satisfied!

Atomicity

```
C atomic-increment

{
    atomic_t v = ATOMIC_INIT(0);
}

P0(atomic_t *v)
{
    int r0;

    r0 = atomic_inc_return_relaxed(v);
}
```

```
P1(atomic_t *v)
{
    int r0;

    r0 = atomic_inc_return_relaxed(v);
}

exists (~v=2)
```

This ‘exists’ clause can NOT be satisfied!

amarula solutions

Mappings to processors

	x86	powerpc	arm64	riscv
`READ_ONCE()`	`mov`	`ldw`	`ldr`	`lw`
`smp_load_acquire()`	`mov`	`ldw; lwsync`	`ldar`	`lw; fence r,rw`
`smp_store_release()`	`mov`	`lwsync; stw`	`stlr`	`fence rw,w; sw`
`smp_mb()`	`lock; addl`	`sync`	`dmb ish`	`fence rw,rw`
`atomic_inc()`	`lock; incl`	LL/SC	`stadd`	`amoadd.w`

> I'm now seeing 1% difference between the runs with 0.3% noise for either of them, [...] I still think that is significant
>
> Will Deacon, on ldr vs. ldar in rcu_dereference()

> 18-32% slower, or 23-47 cycles. [...] So although this test is not a real workload it is a proxy for something people do complain to us about.
>
> Michael Ellerman, on lwsync vs. sync in spin_unlock()

amarula solutions

Concluding remarks

‘the minimum functionality...’ we can rely on?

```
unsigned long __xchg_u32(volatile u32 *ptr, u32 new)
{
   [...] spin_lock_irqsave(ATOMIC_HASH(ptr), flags);
   prev = *ptr; *ptr = new; [...]
```

(from arch/sparc/lib/atomic32.c)

In fact, a recent bug (since fixed) caused GCC to incorrectly use this optimization in a volatile store. In the absence of such bugs, use of WRITE_ONCE() prevents store tearing [...]

(from Documentation/memory-barriers.txt)

("lightweight sync") The memory barrier provides an ordering function for the storage accesses caused by Load, Store, and dcbz instructions [...] in storage that is [...]

(from Power ISA Version 3.0B - Sect. 4.6.3, p. 873)

On all versions of the Cortex-A9 MPCore processor [...] successive reads from the same location [...] can result in the read values not appearing in program order.

(from Read-after-Read Hazards - ARM Ref. 761319)

amarula solutions

‘a guide as to how to use...’ memory barriers?

- Did you consider locking, RCU, . . . ?
- Write a litmus test, or try to
- Cc: these people. . .

We want to hear from you!

Maintainers: Alan Stern, Andrea Parri, Will Deacon, Peter Zijlstra, Boqun Feng, Nicholas Piggin, David Howells, Jade Alglave, Luc Maranget, and Paul E. McKenney

Reviewers: Akira Yokosawa and Daniel Lustig

Email lists: `linux-kernel@vger.kernel.org`, `linux-arch@vger.kernel.org`

amarula solutions

The perfect memory-barrier comment?

```
/* Guarantees that we have nice foo's. */
smp_store_release(&foo->flag, new_flag);
```

amarula solutions

The perfect memory-barrier comment?

```
WRITE_ONCE(foo->data, new_data);    /* A */
[...]

/*
 * Guarantees that we have nice foo's.
 *
 * Orders (A) before (B).
 */
smp_store_release(&foo->flag, new_flag);    /* B */
```

amarula solutions

The perfect memory-barrier comment?

```
WRITE_ONCE(foo->data, new_data);   /* A */
[...]

/*
 * Guarantees that we have nice foo's.
 *
 * Orders (A) before (B).  Matches the smp_load_acquire()
 * in consumer() that orders (C) before (D).
 */
smp_store_release(&foo->flag, new_flag);   /* B */
```

amarula solutions

The perfect memory-barrier comment?

```
WRITE_ONCE(foo->data, new_data);   /* A */
[...]

/*
 * Guarantees that we have nice foo's.
 *
 * Orders (A) before (B).  Matches the smp_load_acquire()
 * in consumer() that orders (C) before (D).
 *
 * Forbids: (C) reads-from (B) AND (A) overwrites (D).
 */
smp_store_release(&foo->flag, new_flag);   /* B */
```

amarula solutions

The perfect memory-barrier comment?

```
WRITE_ONCE(foo->data, new_data);   /* A */
[...]

/*
 * Guarantees that we have nice foo's.
 *
 * Orders (A) before (B).  Matches the smp_load_acquire()
 * in consumer() that orders (C) before (D).
 *
 * Forbids: (C) reads-from (B) AND (A) overwrites (D).
 */
smp_store_release(&foo->flag, new_flag);   /* B */
```

amarula solutions

What's next?

- SRCU
- Data races?
- Mixed-size accesses?

Thanks!

Faster crap is still crap.

Ingo Molnar

Golden rule #12: When the comments do not match the code, they probably are both wrong ;)

Steven Rostedt