Linux-Kernel Memory Model

ISO/IEC JTC1 SC22 WG21 P0124R8 - 2023-08-02 (Informational)

Paul E. McKenney, paulmck@kernel.org
Ulrich Weigand, Ulrich.Weigand@de.ibm.com
Andrea Parri, parri.andrea@gmail.com
Boqun Feng, boqun.feng@gmail.com
Alan Stern, stern@rowland.harvard.edu

History

This is a revision of P0124R7, updated to add the read-copy-update aspects of the Linux-kernel memory model (LKMM) based on the acceptance of RCU into C++26. P0124R7 is a revision of P0124R6, updated based on changes to the Linux-kernel memory model, which has added exclusive locking. This update also fixes a few typos. P0124R6 is in turn a revision of P0124R5, updated with a discussion of `volatile` and of a couple of aspects of the standard that have some relation to control dependencies and locking. P0124R5 is in turn a revision of P0124R4, updated based on the recent de-Alpha-ication of the Linux kernel's core code and on the acceptance of LKMM into the Linux kernel, along with several fixes and corrections. The P0124 series is itself a revision of N4444, updated to add Linux-kernel architecture advice and add more commentary on optimizations. N4444, was a revision of N4374, updated to cover the new `READ_ONCE()` and `WRITE_ONCE()` API members. N4374 was itself a revision of N4322, with updates based on subsequent discussions. This revision adds references to litmus tests in userspace RCU, a paragraph stating goals, and a section discussing the relationship between volatile atomics and loop unrolling.

Introduction

The Linux-kernel memory model has historically been defined very informally in the memory-barriers.txt, atomic_ops.rst, atomic_bitops.txt, atomic_t.txt, and refcount-vs-atomic.rst files in the source tree. Although these five files appear to have been reasonably effective at helping kernel hackers understand what is and is not permitted, they are not necessarily sufficient for non-Linux-kernel C and C++ hackers to derive the corresponding formal model. This document is an attempt to bridge this gap. Up-to-date versions of the formal Linux-kernel memory model may be found in the Linux kernel at `git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git` in the directory `tools/memory-model`, with installation instructions referenced in the `README` file. An earlier version of this model is available from a git archive (`https://github.com/aparri/memory-model`), with installation instructions provided here. This model is a successor to the one described in the two-part LWN series here and here, which is in turn an elaboration of the model described here (video, extended presentation, litmus-test repository). A formal publication on this model may be found here.

This paper is for informational purposes. The hope is that this document will help the C and C++ standard committees understand the existing practice and the constraints from the Linux kernel, and also that it will help the Linux community evaluate which portions of the C11 and C++11 memory models might be useful in the Linux kernel.

All that said, the Linux kernel does not mark declarations of variables and structure fields that are to be manipulated atomically. This means that the `__atomic` built-ins, which in gcc generate the same code as their C11 atomic counterparts, are easier to apply to the Linux kernel than are the C11 atomics.

Variable Access

Loads from and stores to shared (but non-atomic) variables should be protected with the `READ_ONCE()`, `WRITE_ONCE()`, and the now-obsolete `ACCESS_ONCE()` macros, for example:

```
r1 = READ_ONCE(x);
WRITE_ONCE(y, 1);
r2 = ACCESS_ONCE(x); /* Obsolete. */
ACCESS_ONCE(y) = 1;  /* Obsolete. */
```

A `READ_ONCE()`, `WRITE_ONCE()`, and the now-obsolete `ACCESS_ONCE()` accesses may be modeled as a `volatile` `memory_order_relaxed` access. However, please note that these macros are defined to work properly only for properly aligned machine-word-sized variables. Applying `ACCESS_ONCE()` to a large array or structure is unlikely to do anything useful, and use of `READ_ONCE()` and `WRITE_ONCE()` in this situation can result in load-tearing and store-tearing, respectively. Nevertheless, this is their definition. Linux-kernel developers would most certainly not be thankful to the compiler for adding locks to `READ_ONCE()`, `WRITE_ONCE()`, or `ACCESS_ONCE()` when applied to oversized objects. And there has been Linux-kernel use of `READ_ONCE()` and `WRITE_ONCE()` to 64-bit variables on 32-bit systems with the expectation that the compiler would emit a pair of 32-bit accesses, but otherwise respect volatility.

Note that the `volatile` is absolutely required: Non-`volatile` `memory_order_relaxed` is not sufficient. To see this, consider that `READ_ONCE()` can be used to prevent concurrently modified accesses from being hoisted out of a loop or out of unrolled instances of a loop. For example, given this loop:

```
        while (tmp = atomic_load_explicit(a, memory_order_relaxed))
                do_something_with(tmp);
```

The compiler would be permitted to unroll it as follows:

```
        while (tmp = atomic_load_explicit(a, memory_order_relaxed))
                do_something_with(tmp);
                do_something_with(tmp);
                do_something_with(tmp);
                do_something_with(tmp);
        }
```

This would be unacceptable for real-time applications, which need the value to be reloaded from `a` on each iteration, unrolled or not. The `volatile` qualifier prevents this transformation. For example, consider the following loop:

```
        while (tmp = READ_ONCE(a))
                do_something_with(tmp);
```

This loop could still be unrolled, but the read would also need to be unrolled, for example, like this:

```
        for (;;) {
                if (!(tmp = READ_ONCE(a)))
                        break;
                do_something_with(tmp);
                if (!(tmp = READ_ONCE(a)))
                        break;
                do_something_with(tmp);
                if (!(tmp = READ_ONCE(a)))
```

```
                        break;
                do_something_with(tmp);
                if (!(tmp = READ_ONCE(a)))
                        break;
                do_something_with(tmp);
        }
```

Note that use of the new `READ_ONCE()` and `WRITE_ONCE()` macros are recommended for new code, in fact, `ACCESS_ONCE()` has been phased out as of v4.15. Of course, this phase-out has the advantage that `READ_ONCE()` and `WRITE_ONCE()` are a better match for the C and C++ `memory_order_relaxed` loads and stores, give or take volatility.

This raises the question of what exactly the standard guarantees for `volatile`. A more detailed exposition on `volatile` is said to be in preparation, and should that ever emerge from the shadows, this paper will defer to it. In the meantime, referring to N4762:

1. 4.4.1p6.1 says “Accesses through volatile glvalues are evaluated strictly according to the rules of the abstract machine.”
2. 6.8.1p7 states that volatile accesses are side effects.
3. 6.8.2.1p21 calls out volatile accesses as one of the four forward-progress indicators.
4. 9.1.7.2p5 states that the semantics of an access through a volatile glvalue are implementation-defined. Which should not be a surprise to anyone who does not expect the MMIO registers of every device to be ensconced into the standard.
5. 9.1.7.2p6 (a non-normative note) states:

    > volatile is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. Furthermore, for some implementations, volatile might indicate that special hardware instructions are required to access the object. See 6.8.1 for detailed semantics. In general, the semantics of volatile are intended to be the same in C ++ as they are in C.

It is hard to imagine someone intuiting the required semantics of volatile based on the above wording. However, one helpful guideline is that device drivers must work correctly, resulting in the following constraints:

1. Implementations are forbidden from tearing an aligned volatile access when machine instructions of the access's size and type are available. (Note that this intentionally leaves unspecified what to do with 128-bit loads and stores on CPUs having 128-bit CAS but not 128-bit loads and stores.) Concurrent code relies on this constraint to avoid unnecessary load and store tearing.
2. Implementations must not assume anything about the semantics of a volatile access, nor, for any volatile access that returns a value, about the possible set of values that might be returned. (This is strongly implied by the implementation-defined semantics called out above.) Concurrent code relies on this constraint to avoid optimizations that are inapplicable given that other processors might be concurrently accessing the location in question.
3. Aligned machine-sized non-mixed-size volatile accesses interact naturally with volatile assembly-code sequences before and after. This is necessary because some devices must be accessed using a combination of volatile MMIO accesses and special-purpose assembly-language instructions. Concurrent code relies on this constraint in order to achieve the desired ordering properties from combinations of volatile accesses and memory-barrier instructions.

Concurrent code also relies on the first two constraints to avoid undefined behavior that could result due to data races if any of the accesses to a given object was either non-atomic or non-volatile, assuming that all accesses are aligned and machine-sized. The semantics of mixed-size accesses to the same locations are more complex, and are outside the current scope of this document.

At one time, `gcc` guaranteed that properly aligned accesses to machine-word-sized variables would be atomic. Although `gcc` no longer documents this guarantee, there is still code in the Linux kernel that relies on it. These accesses could be modeled as non-`volatile` `memory_order_relaxed` accesses.

The Linux kernel provides `atomic_t` and `atomic_long_t` types. These have `atomic_read()` and `atomic_long_read()` operations that provide non-RMW loads from the underlying variable. They also have `atomic_set()` and `atomic_long_set()` operations that provide non-RMW stores into the the underlying variable. These were originally intended for single-threaded initialization-time and cleanup-time accesses to atomic variables, however, they have since been adapted to operate in a manner similar to `memory_order_relaxed` loads and stores.

The `atomic_t` and `atomic_long_t` types have quite a few other operations that are described in the "Atomic Operations" section. These types could potentially be modeled as volatile atomic `int` for `atomic_t` and volatile atomic `long` for `atomic_long_t`, however, anyone using such a strategy could expect great scrutiny of the code generated at initialization time, when there is no possibility of concurrent access. (Implementations that implement aligned machine-word-sized relaxed atomic loads and stores as normal load and store instructions will pass scrutiny, at least assuming that pointers are machine-word-sized.)

An `smp_store_release()` may be modeled as a `volatile` `memory_order_release` store. Similarly, an `smp_load_acquire()` may be modeled as a volatile `memory_order_acquire` load.

```
r1 = smp_load_acquire(x);
smp_store_release(y, 1);
```

Members of the `rcu_dereference()` family can be modeled as `memory_order_consume` loads. Members of this family include: `rcu_dereference()`, `rcu_dereference_bh()`, `rcu_dereference_sched()`, `srcu_dereference()`, and `lockless_dereference()`. However, `rcu_dereference()` should be representative for litmus-test purposes, at least initially. Similarly, `rcu_assign_pointer()` can be modeled as a `memory_order_release` store. (Exception: when `rcu_assign_pointer()` stores a `NULL` or a pointer to constant data, for example, compile-time initialized data, then `rcu_assign_pointer()` may be modeled as a `memory_order_relaxed` store.)

The `smp_store_mb()` function (`set_mb()` prior to v4.2) assigns the specified value to the specified variable, then executes a full memory barrier, which is described in the next section. This isn't as strong as a `memory_order_seq_cst` store because the following code fragment does not guarantee that the stores to `x` and `y` will be ordered.

```
smp_store_release(x, 1);
smp_store_mb(y, 1);
```

That said, `smp_store_mb()` provides exactly the ordering required for manipulating task state, which is the job for which it was created.

Memory Barriers

The Linux kernel has a variety of memory barriers:

1. `barrier()`, which can be modeled as an `atomic_signal_fence(memory_order_acq_rel)` or an `atomic_signal_fence(memory_order_seq_cst)`.
2. `smp_mb()`, which does not have a direct C11 or C++11 counterpart. On an ARM, PowerPC, or x86 system, it can be modeled as a full memory-barrier instruction (`dmb`, `sync`, and `mfence`, respectively). On an Itanium system, it can be modeled as an `mf` instruction, but this relies on `gcc` emitting an `ld.acq` for an `READ_ONCE()` and an `st.rel` for an `WRITE_ONCE()`. (Peter Zijlstra of Intel notes that although IA64's reference manual claims instructions with acquire and release semantics, the actual hardware implements only full barriers. See commit e4f9bfb3feae ("ia64: Fix up smp_mb__{before,after}_clear_bit()") for Linux-kernel changes based on this situation. Tony Luck and Fenghua Yu are the IA64 maintainers for the Linux kernel.)
3. `smp_rmb()`, which can be modeled (overly conservatively) as an `atomic_thread_fence(memory_order_acq_rel)`. One difference is that `smp_rmb()` need not order prior accesses against later stores, or prior stores against later accesses. Another difference is that `smp_rmb()` need not provide any sort of transitivity, having (lack of) transitivity properties similar to ARM's or PowerPC's address/control/data dependencies.

4. `smp_wmb()`, which can be modeled (again overly conservatively) as an `atomic_thread_fence(memory_order_acq_rel)`. One difference is that `smp_wmb()` need not order prior loads against later accesses, nor prior accesses against later loads. Similar to `smp_rmb()`, `smp_wmb()` need not provide any sort of transitivity.
5. `smp_read_barrier_depends()`, which is a no-op on all architectures other than Alpha. On Alpha, `smp_read_barrier_depends()` may be modeled as a `atomic_thread_fence(memory_order_acq_rel)` or as a `atomic_thread_fence(memory_order_seq_cst)`. As of v4.16 of the Linux kernel, `READ_ONCE()` and a few other primitives include an `smp_read_barrier_depends()`, which means that `smp_read_barrier_depends()` should not be needed in any other non-Alpha-specific code.
6. `smp_mb__before_atomic()`, which provides a full memory barrier before the immediately following non-value-returning atomic operation.
7. `smp_mb__after_atomic()`, which provides a full memory barrier after the immediately preceding non-value-returning atomic operation. Both `smp_mb__before_atomic()` and `smp_mb__after_atomic()` are described in more detail in the later section on atomic operations.
8. `smp_mb__after_unlock_lock()`, which provides a full memory barrier after the immediately preceding lock operation, but only when paired with a preceding unlock operation by this same thread or a preceding unlock operation on the same lock variable. The use of `smp_mb__after_unlock_lock()` is described in more detail in the section on locking.
9. `smp_mb__after_spinlock()`, which provides full ordering after lock acquisition. The ordering guarantees of `smp_mb__after_spinlock()` are a strict superset of those of `smp_mb__after_unlock_lock()`.

There are some additional memory barriers including `mmiowb()`, however, these cover interactions with memory-mapped I/O, so have no counterpart in C11 and C++11 (which is most likely as it should be for the foreseeable future).

Some use cases for these memory barriers may be found here. These are for the userspace RCU library, so drop the leading `cmm_` to get the corresponding Linux-kernel primitive. For example, the userspace `cmm_smp_mb()` primitive translates to the Linux-kernel `smp_mb()` primitive.

Locking Operations

The Linux kernel features “roach motel” ordering on its locking primitives: Prior operations can be reordered to follow a later acquire, and subsequent operations can be reordered to precede an earlier release. The CPU is permitted to reorder acquire and release operations in this way, but the compiler is not, as compiler-based reordering could result in deadlock.

Note that a release-acquire pair does not necessarily result in a full barrier. To see this consider the following litmus test, with `x` and `y` both initially zero, and locks `l1` and `l3` both initially held by the threads releasing them:

```
Thread 1                      Thread 2
--------                      --------
y = 1;                        x = 1;
spin_unlock(&l1);             spin_unlock(&l3);
spin_lock(&l2);               spin_lock(&l4);
r1 = x;                       r2 = y;

assert(r1 != 0 || r2 != 0);
```

In the above litmus test, the assertion can trigger, meaning that an unlock followed by a lock is not guaranteed to be a full memory barrier. And this is where `smp_mb__after_unlock_lock()` comes in:

```
Thread 1                      Thread 2
--------                      --------
y = 1;                        x = 1;
spin_unlock(&l1);             spin_unlock(&l3);
```

```
spin_lock(&l2);               spin_lock(&l4);
smp_mb__after_unlock_lock();  smp_mb__after_unlock_lock();
r1 = x;                       r2 = y;

assert(r1 != 0 || r2 != 0);
```

In contrast, after addition of `smp_mb__after_unlock_lock()`, the assertion cannot trigger.

The above example showed how `smp_mb__after_unlock_lock()` can cause an unlock-lock sequence in the same thread to act as a full barrier, but it also applies in cases where one thread unlocks and another thread locks the same lock, as shown below:

```
Thread 1            Thread 2                        Thread 3
--------            --------                        --------
y = 1;              spin_lock(&l);                  x = 1;
spin_unlock(&l);    smp_mb__after_unlock_lock();    smp_mb();
                    r1 = y;                         r3 = y;
                    r2 = x;

assert(r1 == 0 || r2 != 0 || r3 != 0);
```

Without the `smp_mb__after_unlock_lock()`, the above assertion can trigger, and with it, it cannot. The fact that it can trigger without might seem strange at first glance, but locks are only guaranteed to give sequentially consistent ordering to their critical sections. If you want an observer thread to see the ordering without holding the lock, you need `smp_mb__after_unlock_lock()`.

However, the Linux kernel memory model makes a special case for unlock-lock pairs that access the same lock variable. Even without `smp_mb__after_unlock_lock()`, they are required to provide ordering essentially equivalent to TSO. That is, loads preceding the unlock operation are ordered before all accesses following the lock operation, and stores preceding the unlock are ordered before stores following the lock. (The preceding example does not fall into either category, as it involves ordering a store before a load; hence the `smp_mb__after_unlock_lock()` is required as shown.) The litmus test below illustrates load-load ordering:

```
Thread 1            Thread 2
--------            --------
r1 = x;             y = 1;
spin_unlock(&l);    smp_mb();
spin_lock(&l);      x = 1;
r2 = y;

assert(r1 == 0 || r2 != 0);
```

and the following illustrates store-store ordering:

```
Thread 1            Thread 2            Thread 3
--------            --------            --------
x = 1;              spin_lock(&l);      r2 = y;
spin_unlock(&l);    r1 = x;             smp_mb();
                    y = 1;              r3 = x;

assert(r1 == 0 || r2 == 0 || r3 != 0);
```

The assertions in these two tests will never trigger. This guarantee applies only to unlock and lock operations, not to ordinary release and acquire operations such as `smp_store_release()` or `smp_load_acquire()`.

The `smp_mb__after_spinlock()` barrier is similar to `smp_mb__after_unlock_lock()`, but also guarantees to order accesses preceding the lock acquisition. Only PowerPC needs a non-empty `smp_mb__after_unlock_lock()`, but a non-empty `smp_mb__after_spinlock()` is required by PowerPC, ARMv8, and RISC-V.

The Linux kernel has an embarrassingly large number of locking primitives, but `spin_lock()` and `spin_unlock()` should be representative for litmus-test purposes, at least initially.

Interestingly enough, the Linux kernel's locking operations can be argued to be weaker than those of C11. This argument is based on interpretation of 29.3p3 of the C++11 standard, which states in a non-normative note:

> Although it is not explicitly required that S include locks, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the “happens before” ordering.

That said, current C11 formal memory models specify locking primitives to have roughly the same strength as those of the Linux kernel, based on the following two litmus tests:

```
int main()
{
    atomic_int x = 0;
    atomic_int y = 0;
    mutex m1;
    mutex m2;
    mutex m3;
    mutex m4;

    {{{ { m1.lock();
          y.store(1, memory_order_relaxed);
          m1.unlock();
          m2.lock();
          r1 = x.load(memory_order_relaxed);
          m2.unlock(); }

    ||| { m3.lock();
          x.store(1, memory_order_relaxed);
          m3.unlock();
          m4.lock();
          r2 = y.load(memory_order_relaxed);
          m4.unlock(); }
    }}};

    return 0;
}
```

This first litmus test can observe both `r1` and `r2` equal to zero, which would be prohibited if locking primitives enforced SC behavior.

The second litmus test substitutes a fence for one of the unlock-lock pairs:

```
int main()
{
    atomic_int x = 0;
    atomic_int y = 0;
    mutex mtx;

    {{{ { mtx.lock();
          x.store(1, memory_order_relaxed);
          mtx.unlock();
          mtx.lock();
          r0 = y.load(memory_order_relaxed);
          mtx.unlock(); }

    ||| { y.store(1, memory_order_relaxed);
          atomic_thread_fence(memory_order_seq_cst);
          r1 = x.load(memory_order_relaxed); }
    }}};

    return 0;
}
```

This second litmus test can also observe both `r1` and `r2` equal to zero, which again would be prohibited if locking primitives enforced SC behavior.

Atomic Operations

Atomic operations have three sets of operations, those that are defined on `atomic_t`, those that are defined on `atomic_long_t`, and those that are defined on aligned machine-sized variables, currently restricted to `int` and `long`. However, in the near term, it should be acceptable to focus on a small subset of these operations.

Variables of type `atomic_t` may be stored to using `atomic_set()` and variables of type `atomic_long_t` may be stored to using `atomic_long_set()`. Similarly, variables of these types may be loaded from using `atomic_read()` and `atomic_long_read()`. The historical definition of these primitives has lacked any sort of concurrency-safe semantics, so the user is responsible for ensuring that these primitives are not used concurrently in a conflicting manner.

That said, many architectures treat `atomic_read()` and `atomic_long_read()` as `volatile memory_order_relaxed` loads and a few architectures treat `atomic_set()` and `atomic_long_set()` as `memory_order_relaxed` stores. There is therefore some chance that concurrent conflicting accesses will be allowed at some point in the future, at which point their semantics will be those of `volatile memory_order_relaxed` accesses. However, as noted earlier, any attempt to implement `atomic_t` and `atomic_long_t` as volatile atomic `int` and `long` can expect great scrutiny of the code generated in cases such as initialization where no concurrent accesses are possible.

The remaining atomic operations are divided into those that return a value and those that do not. The atomic operations that do not return a value are similar to C11 atomic `memory_order_relaxed` operations. However, the Linux-kernel atomic operations that do return a value cannot be implemented in terms of the C11 atomic operations. These operations can instead be modeled as `memory_order_relaxed` operations that are both preceded and followed by the Linux-kernel `smp_mb()` full memory barrier, which is implemented using the `DMB` instruction on ARM and the `sync` instruction on PowerPC. Alternatively, if appropriate, `smp_mb__before_atomic()` and `smp_mb__after_atomic()` could be used in place of `smp_mb()`. Note that in the case of the CAS operations `atomic_cmpxchg()`, `atomic_long_cmpxchg`, and `cmpxchg()`, the full barriers are required only in the success case as v4.3 (before that, full barriers were required in both cases). Strong memory ordering can be added to the non-value-returning atomic operations using `smp_mb__before_atomic()` before and/or `smp_mb__after_atomic()` after.

For some of the value-returning atomic operations, there are also sets of variants introduced in v4.4. These variants use suffixes to indicate their ordering guarantees. There are three kinds of variants: `_relaxed`, `_acquire` and `_release`, and they are similar to the corresponding C11 `memory_order_relaxed`, `memory_order_acquire` and `memory_order_release` atomic operations, except that they are volatile, which means they won't be optimized out or merged with other atomic operations. Note that in the case of the variants of CAS operations `atomic_cmpxchg()`, `atomic_long_cmpxchg`, and `cmpxchg()`, the ordering guarantees are required only in the success case.

Note that C11 compilers are within their rights to assume data-race freedom when determining what optimizations to carry out. This will break the still-common Linux-kernel practice of assuming relaxed semantics for normal accesses to non-atomic variables, hence the suggestions to disable code-motion optimizations across atomics using full barriers and/or Linux-kernel `barrier()` macros.

The operations are summarized in the following table. An initial implementation of a tool could start with `atomic_add()`, `atomic_sub()`, `atomic_xchg()`, and `atomic_cmpxchg()`.

Operation Class	**int**	**long**

Add/Subtract	`void atomic_add(int i, atomic_t *v)`
	`void atomic_sub(int i, atomic_t *v)`
	`void atomic_inc(atomic_t *v)`
	`void atomic_dec(atomic_t *v)`
		`void atomic_long_add(long i, atomic_long_t *v)`
		`void atomic_long_sub(long i, atomic_long_t *v)`
		`void atomic_long_inc(atomic_long_t *v)`
		`void atomic_long_dec(atomic_long_t *v)`

Add/Subtract, Value Returning, (Variants Available)	`int atomic_inc_return(atomic_t *v)`
	`int atomic_dec_return(atomic_t *v)`
	`int atomic_add_return(int i, atomic_t *v)`
	`int atomic_sub_return(int i, atomic_t *v)`
		`long atomic_long_inc_return(atomic_long_t *v)`
		`long atomic_long_dec_return(atomic_long_t *v)`
		`long atomic_long_add_return(long i, atomic_long_t *v)`
		`long atomic_long_sub_return(long i, atomic_long_t *v)`

Add/Subtract, Value Returning, (No Variants)	`int atomic_inc_and_test(atomic_t *v)`
	`int atomic_dec_and_test(atomic_t *v)`
	`int atomic_sub_and_test(int i, atomic_t *v)`
	`int atomic_add_negative(int i, atomic_t *v)`
		`long atomic_long_inc_and_test(atomic_long_t *v)`
		`long atomic_long_dec_and_test(atomic_long_t *v)`
		`long atomic_long_sub_and_test(long i, atomic_long_t *v)`
		`long atomic_long_add_negative(long i, atomic_long_t *v)`

Exchange, (Variants Available)	`int atomic_xchg(atomic_t *v, int new)`
	`int atomic_cmpxchg(atomic_t *v, int old, int new)`
		`long atomic_long_xchg(atomic_long_t *v, long new)`
		`long atomic_long_cmpxchg(atomic_code_t *v, long old, long new)`

Conditional Add/Subtract	`int atomic_add_unless(atomic_t *v, int a, int u)`
	`int atomic_inc_not_zero(atomic_t *v)`
		`long atomic_long_add_unless(atomic_long_t *v, long a, long u)`
		`long atomic_long_inc_not_zero(atomic_long_t *v)`

Bit Test/Set/Clear (Generic)	`void set_bit(unsigned long nr, volatile unsigned long *addr)`
	`void clear_bit(unsigned long nr, volatile unsigned long *addr)`
	`void change_bit(unsigned long nr, volatile unsigned long *addr)`	

Bit Test/Set/Clear, Value Returning (Generic, No Variants)	`int test_and_set_bit(unsigned long nr, volatile unsigned long *addr)`
	`int _atomic_dec_and_lock(atomic_t *atomic, spinlock_t *lock)`
	`int test_and_clear_bit(unsigned long nr, volatile unsigned long *addr)`
	`int test_and_change_bit(unsigned long nr, volatile unsigned long *addr)`	

Lock-Barrier Operations (Generic)	`int test_and_set_bit_lock(unsigned long nr, unsigned long *addr)`
	`void clear_bit_unlock(unsigned long nr, unsigned long *addr)`
	`void __clear_bit_unlock(unsigned long nr, unsigned long *addr)`	

Exchange (Generic, Variants Available)	`T *xchg(T *v, new)`
	`T *cmpxchg(T *v, T old, T new)`

The rows marked “(Generic)” are type-generic, applying to any aligned machine-word-sized quantity supported by all architectures that the Linux kernel runs on. The set of types is currently those of size `int` and those of size `long`. The “Lock-Barrier Operations” have `memory_order_acquire` semantics for `test_and_set_bit_lock()` and `_atomic_dec_and_lock()`, and have `memory_order_release` for the other primitives. Otherwise, the usual Linux-kernel rule holds: If no value is returned, `memory_order_relaxed` semantics apply, otherwise the operations behave as if there was `smp_mb()` before and after. And for those value-returning primitives, the rows marked “(Variants Available)” have `_relaxed`/`_acquire`/`_release` variants, whereas the rows marked “(No Variants)” don't.

The following table gives rough C11 counterparts for the Linux-kernel atomic operations called out above:

Linux-Kernel Operation	C11 Counterpart
Add/Subtract	

`void atomic_add(int i, atomic_t *v)`
`void atomic_long_add(long i, atomic_long_t *v)`
	`atomic_fetch_add_explicit(v, i, memory_order_relaxed)`

`void atomic_sub(int i, atomic_t *v)`
`void atomic_long_sub(long i, atomic_long_t *v)`
	`atomic_fetch_sub_explicit(v, i, memory_order_relaxed)`

`void atomic_inc(atomic_t *v)`
`void atomic_long_inc(atomic_long_t *v)`
	`atomic_fetch_add_explicit(v, 1, memory_order_relaxed)`

`void atomic_dec(atomic_t *v)`
`void atomic_long_dec(atomic_long_t *v)`
	`atomic_fetch_sub_explicit(v, 1, memory_order_relaxed)`

Add/Subtract, Value Returning (Variants Available)	

`int atomic_inc_return(atomic_t *v)`
`long atomic_long_inc_return(atomic_long_t *v)`
	`atomic_fetch_add(v, 1) - 1`

`int atomic_dec_return(atomic_t *v)`
`long atomic_long_dec_return(atomic_long_t *v)`
	`atomic_fetch_sub(v, 1) + 1`

`int atomic_add_return(int i, atomic_t *v)`
`long atomic_long_add_return(long i, atomic_long_t *v)`
	`atomic_fetch_add(v, i) - i`

`int atomic_sub_return(int i, atomic_t *v)`
`long atomic_long_sub_return(long i, atomic_long_t *v)`
	`atomic_fetch_sub(v, i) + i`

Value Returning, No Variants	

`int atomic_inc_and_test(atomic_t *v)`
`long atomic_long_inc_and_test(atomic_long_t *v)`
	`atomic_fetch_add(v, 1) == -1`

`int atomic_dec_and_test(atomic_t *v)`
`long atomic_long_dec_and_test(atomic_long_t *v)`
	`atomic_fetch_sub(v, 1) == 1`

`int atomic_sub_and_test(int i, atomic_t *v)`
`long atomic_long_sub_and_test(long i, atomic_long_t *v)`
	`atomic_fetch_sub(v, i) == i`

`int atomic_add_negative(int i, atomic_t *v)`
`long atomic_long_add_negative(long i, atomic_long_t *v)`
	`atomic_fetch_add(v, i) < 0`

Exchange, Variants Available	

`int atomic_xchg(atomic_t *v, int new)`
`long atomic_long_xchg(atomic_long_t *v, long new)`
`T *xchg(T *v, new)`
	`atomic_exchange(v, new)`

`int atomic_cmpxchg(atomic_t *v, int old, int new)`
`long atomic_long_cmpxchg(atomic_code_t *v, long old, long new)`
`T *cmpxchg(T *v, T old, T new)`
	`t = old;`
	`atomic_compare_exchange_strong_explicit(v, &t, new, memory_order_seq_cst, memory_order_relaxed)`
	`return t;`

Conditional Add/Subtract	

`int atomic_add_unless(atomic_t *v, int a, int u)`
`long atomic_long_add_unless(atomic_long_t *v, long a, long u)`
	`t = atomic_load_explicit(v, memory_order_relaxed);`
	`do {`
	`  if (t == u) return false;`
	`} while (!atomic_compare_exchange_weak_explicit(a, t, t + a, memory_order_seq_cst, memory_order_relaxed));`
	`return true;`

`int atomic_inc_not_zero(atomic_t *v)`
`long atomic_long_inc_not_zero(atomic_long_t *v)`
	`t = atomic_load_explicit(v, memory_order_relaxed);`
	`do {`
	`  if (t == 0) return false;`
	`} while (!atomic_compare_exchange_weak_explicit(a, t, t + 1, memory_order_seq_cst, memory_order_relaxed));`
	`return true;`

Bit Test/Set/Clear/Change	

	p and `mask` in below:
	`unsigned long *p = ((unsigned long *)addr) + nr / (sizeof(unsigned long) * CHAR_BIT);`
	`unsigned long mask = 1u << (nr % (sizeof(unsigned long) * CHAR_BIT))`

`void set_bit(unsigned long nr, volatile void *addr)`	`atomic_fetch_or_explicit(p, mask, memory_order_relaxed);`
`void clear_bit(unsigned long nr, volatile void *addr)`	`atomic_fetch_and_explicit(p, ~mask, memory_order_relaxed);`

`void change_bit(unsigned long nr, volatile void *addr)`	`atomic_fetch_xor_explicit(p, mask, memory_order_relaxed);`
`int test_and_set_bit(unsigned long nr, volatile void *addr)`	`return !!(atomic_fetch_or_explicit(p, mask, memory_order_relaxed) & mask);`
`int test_and_clear_bit(unsigned long nr, volatile void *addr)`	`return !!(atomic_fetch_and_explicit(p, ~mask, memory_order_relaxed) & mask);`
`int test_and_change_bit(unsigned long nr, volatile void *addr)`	`return !!(atomic_fetch_xor_explicit(p, mask, memory_order_relaxed) & mask);`
Lock Bit Test and Set/Clear	
	p and `mask` in below: `unsigned long *p = ((unsigned long *)addr) + nr / (sizeof(unsigned long) * CHAR_BIT); unsigned long mask = 1u << (nr % (sizeof(unsigned long) * CHAR_BIT))`
`int test_and_set_bit_lock(unsigned long nr, volatile void *addr)`	`return !!(atomic_fetch_or_explicit(p, mask, memory_order_acquire) & mask);`
`void clear_bit_unlock(unsigned long nr, volatile void *addr)`	`atomic_fetch_and_explicit(p, ~mask, memory_order_release);`

The bit test/set/clear and lock-barrier operations map to atomic bit operations, but accept very large bit numbers. The upper bits of the bit number select the `unsigned long` element of an array, and the lower bits select the bit to operate on within the selected `unsigned long`.

Control Dependencies

The Linux kernel provides a limited notion of control dependencies, ordering prior loads against control-dependent stores in some cases. Extreme care is required to avoid control-dependency-destroying compiler optimizations. The restrictions applying to control dependencies include the following:

1. Control dependencies can order prior loads against later dependent stores, however, they do *not* order prior loads against later dependent loads. (Use `memory_order_consume` or `memory_order_acquire` if you require this behavior.)
2. A load heading up a control dependency must use `READ_ONCE()`. Similarly, the store at the other end of a control dependency must also use `WRITE_ONCE()`. As of v4.16 of the Linux kernel, `READ_ONCE()` can also head address and data dependency chains, which allowed Alpha-specific code to be removed from almost all of the core kernel.
3. If both legs of a given `if` or `switch` statement store the same value to the same variable, then those stores cannot participate in control-dependency ordering.
4. Control dependencies require at least one run-time conditional that depends on the prior load and that precedes the following store.
5. The compiler must perceive both the variable loaded from and the variable stored to as being shared variables. For example, the compiler will not perceive an on-stack variable as being shared unless its address has been taken and exported to some other thread (or alias analysis has otherwise been defeated).
6. Control dependencies are not transitive. In this regard, their behavior is similar to ARM or PowerPC control dependencies.

For more information, see the "CONTROL DEPENDENCIES" of the Linux kernel's infamous memory-barriers.txt documentation.

The C and C++ standards do not guarantee ordering based on control dependencies. Therefore, this list of restriction is subject to change as compilers become increasingly clever and aggressive. Nevertheless, these standards do have some restrictions that are at least somewhat related to control dependencies:

1. The compiler is not permitted to generate data races. In many cases, this can prohibit the compiler from hoisting a normal access out of a conditional.
2. The compiler is not permitted to invent either an atomic store or a volatile access.

Note that these restrictions apply even if the conditional depends only on normal not-atomic/non-volatile accesses. To see this, consider an unadored flag that is set just before `main()` spawns its threads. Code that is called both during pre-spawn initialization and from the threads could load that flag in order to determine whether or not data races are possible, and the compiler would need to honor such checks.

RCU Grace-Period Relationships

The Linux-kernel RCU API is quite extensive. This section focuses on only those aspects of this API that have (sometimes rough) counterparts in the initial C++26 RCU specification.

The publish-subscribe portions of RCU are captured by the combination of `rcu_assign_pointer()`, which can be modeled as a `memory_order_release` store, and of the `rcu_dereference()` family of primitives, which can be modeled as `memory_order_consume` loads, as was noted earlier.

Grace periods can be modeled as described in Appendix D of User-Level Implementations of Read-Copy Update. There are a number of grace-period primitives in the Linux kernel, but `rcu_read_lock()`, `rcu_read_unlock()`, and `synchronize_rcu()` are good places to start. The grace-period relationships can be describe using the following abstract litmus test:

```
Thread 1                    Thread 2
--------                    --------
rcu_read_lock();            S2a;
S1a;                        synchronize_rcu();
S1b;                        S2b;
rcu_read_unlock();
```

If either of `S1a` or `S1b` precedes `S2a`, then both must precede `S2b`. Conversely, if either of `S1a` or `S1b` follows `S2b`, then both must follow `S2a`. Additional litmus tests may be found here and here. For the litmus tests using the userspace RCU library, drop the leading `cmm_` to get the corresponding Linux-kernel primitives.

Given a high-quality implementation of `memory_order_consume`, a willingness to accept `memory_order_acquire` overhead on RCU pointer traversals, or a willingness to adhere to coding guidelines similar to those in rcu_dereference.rst, RCU can be implemented as a library, and is further included in C++26. A rough correspondence between the Linux-kernel and C++26 RCU API is as follows:

Linux Operation	C++26 Implementation
`rcu_head` structure	`std::rcu_obj_base<T>` [1]
	`std::rcu_default_domain()` [2]
`rcu_read_lock()`	`std::scoped_lock l(std::rcu_default_domain())` [3]
`rcu_read_unlock()`	`}` (RCU guard "l" goes out of scope)
`synchronize_rcu()`	`std::rcu_synchronize()`
`call_rcu()`	`.retire()` (`rcu_obj_base` method) [4][5]
`kfree_rcu_mightsleep()`	`std::rcu_retire()` (but also allows “deleter” callback) [6]
`rcu_barrier()`	`std::rcu_barrier()`

Notes:

1. In Linux-kernel C code, the `rcu_head` structure is included into the RCU-protected structure. In C++26, the RCU-protected structure/class inherits from `std::rcu_obj_base`.

2. The Linux kernel does not need a counterpart to `std::rcu_default_domain()`, which in C++26 serves only to provide an argument to `scoped_lock` and friends. However, the Linux does provide SRCU, which provides multiple `srcu_struct` domains.
3. The usual C++ tricks may be used to obtain RCU read-side critical sections that are not confined to a specific scope.
4. See Table 2 in P2545R4: Read-Copy Update for example usage.
5. In the Linux kernel, it is permissible to hold a lock across a call to `call_rcu()` that is also acquired by some callback function passed to some `call_rcu()` instance (including this one). In C++26, doing this with `.retire()` and some deleter passed to some `.retire` instance can deadlock. C++26 imposes this potential restriction in order to allow RCU to be implemented in environments where RCU cannot be permitted to create its own threads. Of course, a C++26 RCU implementation could choose to avoid these deadlocks, but then any C++ program relying on this would be non-portable.
6. The similarity between `kfree_rcu_mightsleep()` and `std::rcu_retire()` is that both are "non-intrusive", that is, a structure passed to `kfree_rcu_mightsleep()` need not contain an `rcu_head` structure and a structure/class passed to `std::rcu_retire()` need not inherit from `std::rcu_obj_base<T>`. Both of these functions might allocate, and both might wait for a synchronous grace period.

The C++26 RCU API can be implemented in terms of the Linux-kernel RCU API, or, more likely, that of the very similar userspace RCU library. An example of the latter may be found in the RCUCPPbindings github repository.

Summary of Differences With Examples

This section looks in more detail at functionality that the Linux kernel provides that is not available from the C11 standard.

1. `ACCESS_ONCE()`
2. `READ_ONCE()`
3. `WRITE_ONCE()`
4. `smp_mb()`
5. `smp_read_barrier_depends()`
6. `Locking Operations`
7. `Value-Returning Atomics`
8. `Control Dependencies`

`ACCESS_ONCE()`

There is no C11 syntax corresponding to `ACCESS_ONCE()`, which enables both loads and stores. However, this problem was solved by the removal of `ACCESS_ONCE()` from v4.15 of the Linux kernel in favor of `READ_ONCE()` and `WRITE_ONCE()`, which are described below.

`READ_ONCE()`

Although the semantics of `READ_ONCE()` are tantalizingly close to those of a C11 `volatile memory_order_relaxed` atomic read, there are important differences:

1. `READ_ONCE()` can head an address, data, or control dependency chain. This is of course more like `memory_order_consume` load, but all known implementations promote such loads to `memory_order_acquire`, which is stronger than needed by `READ_ONCE()`.
2. `READ_ONCE()` is permitted to tear when used on objects too large for the available load instructions. (And yes, there are parts of the Linux kernel that use `READ_ONCE()` in situations where it must tear, and these users would be decidedly unamused by invention and use of locks by the compiler.)

Therefore, `READ_ONCE()` cannot be implemented in terms of C11 atomics.

`WRITE_ONCE()`

The semantics of `WRITE_ONCE()` are also quite close to those of a C11 `volatile` `memory_order_relaxed` atomic store. However, as with `READ_ONCE()`, `WRITE_ONCE()` is permitted to tear when used on objects too large for the available store instructions. (And yes, there are parts of the Linux kernel that use `WRITE_ONCE()` in situations where it must tear, and these users would be decidedly unamused by invention and use of locks by the compiler.)

Therefore, `WRITE_ONCE()` cannot be implemented in terms of C11 atomics.

`smp_mb()`

Quoting 29.3p8 of the C++11 standard:

> Fences cannot, in general, be used to restore sequential consistency for atomic operations with weaker ordering specifications.

In contrast, `smp_mb()` guarantees to restore sequential consistency among accesses that use `READ_ONCE`, `WRITE_ONCE()`, or stronger. For example, the following Linux-kernel code would forbid non-SC outcomes:

```
int x, y, r0, r1, r2, r3;

void thread0(void)
{
  WRITE_ONCE(x, 1);
}

void thread1(void)
{
  WRITE_ONCE(y, 1);
}

void thread2(void)
{
  r0 = READ_ONCE(x);
  smp_mb()
  r1 = READ_ONCE(y);
}

void thread3(void)
{
  r2 = READ_ONCE(y);
  smp_mb()
  r3 = READ_ONCE(x);
}
```

In contrast, the closest C11 analog can permit the non-SC outcomes and still conform to the standard:

```
atomic_int x, y;
int r0, r1, r2, r3;

void thread0(void)
{
  atomic_store_explicit(x, 1, memory_order_relaxed);
}

void thread1(void)
{
  atomic_store_explicit(y, 1, memory_order_relaxed);
}
```

```

void thread2(void)
{
  r0 = atomic_load_explicit(x, memory_order_relaxed);
  atomic_thread_fence(memory_order_seq_cst);
  r1 = atomic_load_explicit(y, memory_order_relaxed);
}

void thread3(void)
{
  r2 = atomic_load_explicit(y, memory_order_relaxed);
  atomic_thread_fence(memory_order_seq_cst);
  r3 = atomic_load_explicit(x, memory_order_relaxed);
}
```

That said, it is not clear that anything in the Linux kernel cares whether or not sequential consistency is restored.

`smp_read_barrier_depends()`

Although it is legal C11 to say “`atomic_thread_fence(memory_order_consume)`”, all known C11 implementations promote this to an acquire fence. Within the Linux kernel, this would have the undesirable effect of promoting `rcu_dereference()` to acquire as well. Linux therefore needs to continue defining `smp_read_barrier_depends()` as `smp_mb()` on DEC Alpha and nothingness elsewhere.

Note again that as of v4.16 of the Linux kernel, `READ_ONCE()` and a few other primitives include `smp_read_barrier_depends()`, which means that `smp_read_barrier_depends()` should not be needed anywhere else in the Linux kernel aside from Alpha-specific code.

Locking Operations

Consider the following litmus test:

```
void thread0(void)
{
  spin_lock(&my_lock);
  WRITE_ONCE(x, 1);
  spin_unlock(&my_lock);
  spin_lock(&my_lock);
  r0 = READ_ONCE(y);
  spin_unlock(&my_lock);
}

void thread1(void)
{
  WRITE_ONCE(y, 1);
  smp_mb();
  r1 = READ_ONCE(x);
}
```

The Linux kernel is currently within its rights to arrive at the non-SC outcome `r0 == 0 && r1 == 0`. This is inconsistent with C11.

Value-Returning Atomics

Linux's value-returning atomics provide unconditional ordering. For example, in the following code fragment, the outcome `r0 == 1 && r1 == 0` is forbidden:

```
int x, y, z, dummy;
int r0 = 42;
int r1 = 43;

void thread0(void)
{
  WRITE_ONCE(x, 1);
  dummy = xchg(&z, 1);
  WRITE_ONCE(y, 1);
}

void thread1(void)
{
  r0 = smp_load_acquire(&y);
  r1 = READ_ONCE(x);
}
```

In contrast, the closest C11 analog (from David Majnemer and analyzed by Chandler Carruth) does not prohibit this outcome:

```
atomic_int x, y, z;
int r0 = 42;
int r1 = 43;

void thread0(void)
{
  atomic_store_explicit(x, 1, memory_order_relaxed);
  atomic_store_explicit(z, 1, memory_order_seq_cst);
  atomic_store_explicit(y, 1, memory_order_relaxed);
}

void thread1(void)
{
  r0 = atomic_load_explicit(y, memory_order_acquire);
  r1 = atomic_load_explicit(x, memory_order_relaxed);
}
```

Note that this category includes non-value-returning atomics enclosed within `smp_mb__before_atomic()`/`smp_mb__after_atomic()` pairs. For example, the Linux-kernel variant of `thread0()` could be written as follows with the same outcome, assuming that `z` is declared as `atomic_t` instead of `int`:

```
void thread0(void)
{
  WRITE_ONCE(x, 1);
  smp_mb__before_atomic();
  atomic_inc(&z);
  smp_mb__after_atomic();
  WRITE_ONCE(y, 1);
}
```

Control Dependencies

The Linux kernel provides control dependencies and C11 does not, so `READ_ONCE()` must either retain its current implementation or must be promoted to acquire. Those wishing to use `READ_ONCE()` to head control-dependency chains are warned that this type of dependency is quite fragile. For more information, see the "CONTROL DEPENDENCIES" of the Linux kernel's infamous memory-barriers.txt documentation.

So You Want Your Arch To Use C11 Atomics...

So suppose that you want your Linux-kernel architecture to use C11 atomics. How should you go about it? This section looks at three scenarios: (1) A new architecture, (2) Partial conversion of an existing architecture, and (3) Full conversion of an existing architecture. Each of these is covered by one of the following sections.

New Architecture

The potential advantages of using the C11 memory model for a new architecture include:

1. Delegating implementation of atomic primitives to the compiler.
2. If multiple architectures take this approach, a reduction in the amount of architecture-specific code.
3. The compiler can undertake more optimizations. It is left to the reader to decide whether this would be an advantage or a disadvantage.

`READ_ONCE()` and `WRITE_ONCE()` should continue to use the existing definitions in order to avoid the C11-mandated use of locking for oversized objects. (As should `ACCESS_ONCE()` in pre-v4.15 kernels.)

Memory barriers could be implemented in terms of the C11 `atomic_signal_fence()` and `atomic_thread_fence()` functions as follows:

Linux Operation	C11 Implementation
`barrier()`	`atomic_signal_fence(memory_order_seq_cst()` (if safe)
`barrier()`	`__asm__ __volatile__("": : :"memory")` (otherwise)
`smp_mb()`	`atomic_thread_fence(memory_order_seq_cst)` (if safe)
`smp_mb()`	Inline assembly otherwise
`smp_rmb()`	`atomic_thread_fence(memory_order_acq_rel()` (if safe and efficient)
`smp_rmb()`	Inline assembly otherwise
`smp_wmb()`	`atomic_thread_fence(memory_order_acq_rel()` (if safe and efficient)
`smp_wmb()`	Inline assembly otherwise
`smp_read_barrier_depends()`	As in the Linux kernel
`smp_mb__after_atomic()` and `smp_mb__before_atomic()`	Depends on implementation of non-value-returning read-modify-write operations
`smp_mb__after_unlock_lock()`	Depends on implementation of locking primitives
`smp_mb__after_spinlock()`	Depends on implementation of locking primitives

The Linux kernel's locking primitives will likely need to remain as hard-coded assembly for some time to come, particularly for the locking primitives that interact with irq or bottom-half environments. Over time, it might well prove that the compiler can generate “good enough” locking primitives, but careful analysis and inspection should be used to make that determination.

The `atomic_t` and `atomic_long_t` types could be implemented as volatile atomic `int` and `long`. However, this would require inspecting the code that the compiler emits to ensure that the value-returning atomic read-modify-write primitives provide full ordering both before and after, as required for the Linux kernel. Because the C11 compiler might perform optimizations that violate the full-ordering requirement (optimizations based on the assumption of data-race freedom being but one example), it would be wise to add `barrier()` directives at the beginnings and ends of the definitions of the value-returning atomic read-modify-write primitives. This prevents the compiler from carrying out any code-motion optimizations across the `barrier()` directive. In addition, because the C11 compiler can in many cases optimize away atomic operations whose results are not used, and their ordering properties with them, it is wise to place `atomic_thread_fence(memory_order_seq_cst)` before and after C11 atomics that are used to implement Linux-kernel value-returning read-modify-write atomics. Note that this assumes that the implementation in question provides `atomic_thread_fence(memory_order_seq_cst)` ordering properties compatible with `smp_mb()`.

The generic atomic operations might be implemented by casting to volatile atomic objects, and, failing that, inline assembly as is currently used in the Linux kernel.

Until such time as the C11 memory model implements control dependencies, the Linux kernel must implement them. Similarly, RCU must currently also be implemented by the Linux kernel rather than the compiler. That said, TSO machines (for example, x86 and the mainframe) can use volatile `memory_order_consume` loads to implement the `rcu_dereference()` family of primitives without incurring performance penalties.

Partial Conversion of Existing Architecture

In some sense, a new architecture has less to lose by letting the compiler have a go at implementing atomic operations and memory barriers. In contrast, an existing architecture likely already has well-tested high-performance primitives implemented with inline assembly. Not only that, existing architectures might need to support older compilers that do not have robust implementations of C11 atomics. Therefore, any change to C11 should be implemented cautiously, if at all. One way of proceeding cautiously is to do a partial conversion, preferably permitting easy fallback to the original inline assembly. The non-value-returning read-modify-write atomics are likely the safest and easiest C11 primitives to start with.

Full Conversion of Existing Architecture

Full conversion of an existing architecture to C11 requires even more bravery, to say nothing of more complete validation of the relevant C11 functions. For example, it would be wise to provide a Kconfig option selecting between the existing inline assembly and the C11 atomics. This would permit continued use of old compilers where needed, and also allow users to decide when they are ready to trust C11. It would also be wise to look into David Howells's experimental conversion of the Linux kernel x86 architecture to C11 atomics.

Summary

This document makes a first attempt to present a formalizable model of the Linux kernel memory model, including variable access, memory barriers, locking operations, atomic operations, control dependencies, and RCU grace-period relationships. The general approach is to reduce the kernel's memory model to some aspect of memory models that have already been formalized, in particular to those of C11, C++11, ARM, and PowerPC. A formal Linux-kernel memory model has been accepted into the mainline Linux kernel as of April 2, 2018 in the `tools/memory-model` directory, and the corresponding paper has also been presented at ASPLOS 2018.