WG21/P0868R1: Selected RCU Litmus Tests

Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@linux.vnet.ibm.com

Alan Stern
The Rowland Institute at Harvard University
stern@rowland.harvard.edu

Andrew Hunter
Google
ahh@google.com

Jade Alglave
University College London and Microsoft Research
j.alglave@ucl.ac.uk

Luc Maranget
INRIA Paris
luc.maranget@inria.fr

The indefatigable "TBD"

November 20, 2017

Abstract

This document provides a set of litmus tests for read-copy update (RCU) that are selected to help work out ordering constraints and requirements. All of these litmus tests illustrate patterns that a correct RCU implementation must prohibit, although a few of them are closely related to litmus tests that can be allowed. These litmus tests use a C-language syntax similar to that of the Linux kernel because we do not yet have an executable C++ memory model that includes RCU.

1 Introduction

This document assumes that the reader has some knowledge of RCU, for example, as provided by WG21/P0461R2 [6], WG21/P0297R1 [5], WG21/P0232R0 [8], and WG21/P0561R1 [7]. An good understanding of RCU read-side critical sections (`rcu_reader`) and RCU grace periods (for example, `synchronize_rcu()`) will be particularly helpful. Some knowledge of memory ordering and related tools is also helpful [1, 2].

For the tl;dr crowd, here is a rough summary of the relevant RCU rules, where an RCU read-side critical section is the lifetime of an `rcu_reader` and where an RCU grace period is the duration of a call to `synchronize_rcu()`:

1. If any part of a given RCU read-side critical section precedes anything preceding a given RCU grace period, then that entire RCU read-side critical section, along with everything preceding it, must precede anything following that RCU grace period.

2. If anything following a given RCU grace period precedes any part of a given RCU read-side critical section, then anything preceding that RCU grace period must precede that entire RCU read-side critical section, along with everything following it.

3. It is permissible for a given RCU grace period to completely overlap a given RCU read-side critical section, so that the grace period begins before the critical section begins and ends after the critical section ends. However, the reverse is absolutely forbidden as a consequence of the previous pair of rules.

4. RCU read-side critical sections are not required to impose any ordering other than that specified by the above rules. In particular, if a program contains no RCU grace periods, the RCU read-side critical sections need not have any effect whatsoever.

5. In the absence of RCU read-side critical sections, RCU grace periods have the same ordering properties as do full memory fence. However, if a given execution is forbidden in the absence of RCU read-side critical sections, adding such critical sections cannot cause that execution to become allowed. The C++ full memory fence is `atomic_thread_fence(memory_order_seq_cst)`.

The goal is to arrive at wording that causes the C++ standard to enforce these rules.

Section 2 gives an overview of litmus-test syntax and semantics, Section 3 presents a series of of litmus tests intended to illustrate ordering properties, Section 4 provides a rationale for the various ordering properties, Section 6 presents a prototype C11 memory model that includes RCU, and finally, Section 7 provides a decoder ring for the otherwise inexplicable litmus-test filenames.

2 A Tour Through A Litmus Test

This section takes a tour through Listing 2.1, a realistic RCU litmus test that is nevertheless not useful for semantics discussion due to its reliance on the infamous `memory_order_consume` load. However, this litmus test does represent the bread-and-butter use case within the Linux kernel, so it is a worthwhile illustration of litmus-test syntax and semantics.

Line 1 identifies the language (“`C`”, which includes C11, as opposed as some other language or some CPU’s assembly language) and names the test. As far as the litmus-test analysis tools are concerned, the name is arbitrary, but these tests use a convention that identifies the type of the test. This convention is explained in Section 7. There are other conventions: In some situations, a more concise but more specialized naming scheme is desirable, and in other situations the naming scheme is automatically generated by one of many tools and scripts.

Lines 2-4 contain initialization statements. All variables are by default initialized to zero, so in cases zero initialization is sufficient, line 3 could be omitted. However, lines 2 and 4 are required. There are two initialization statements on line 3. The first statement (`0:r3=z0`) specifies that process 0’s local register `r3` is to be initialized to

Listing 2.1: Classic RCU Use Case

```
1 C MP+o-sr-r+rlk-o-addr-o-rulk
2 {
3  int *0:r3=z0; atomic_int *x0=y0;
4 }
5
6 P0(atomic_int *x0, atomic_int *y0)
7 {
8   atomic_store_explicit(x0, r3, memory_order_relaxed);
9   synchronize_rcu();          /* std::synchronize_rcu(); */
10  y0 = 1;
11 }
12
13
14 P1(atomic_int *x0)
15 {
16  int *r1;
17  int r2;
18
19  rcu_read_lock();            /* std::rcu_reader rr; */
20  r1 = atomic_load_explicit(x0, memory_order_consume);
21  r2 = *r1;
22  rcu_read_unlock();
23 }
24
25
26 exists
27 (1:r1=y0 /\ 1:r2=1)
```

the address of a global variable named `z0`. Because `z0` isn't otherwise mentioned in the initialization section, its initial value is zero. The second statement (`x0=y0`) specifies that global variable `x0` is initialized to the address of another zero-initialized global variable `y0`.

Variables not requiring initialization need not be declared, which raises the question of how global and local variables are distinguished. The answer evokes old memories of FORTRAN: Variables starting with "r" are local, and all others are global.[1]

Lines 6-11 and 14-23 define processes `P0()` and `P1()`, respectively. Processes can be arbitrarily named, as long as the first letter is `P` and the remaining letters are numeric, and numbered consecutively from zero.

A given process can directly access only those global variables passed in by reference. Therefore, line 6 shows that `P0` can directly access only `x0` and `y0`. Similarly, line 14 shows that `P1` can directly access only `x0`, however, it might indirectly access either `y0` or `z0` because the addresses of these two variables might be contained in `x0`.

Processes of course contain statements, and the current versions of litmus-test analysis tools [1, 2] only handle Linux-kernel statements for the various RCU-related primitives. As a service to the C++-savvy reader, close C++ equivalents are provided as comments. The tool understands ordering, so for example, the tool understands that the effects of lines 20 and 21 might become visible in either order, and in fact might appear in different orders to different processes.

These processes should be viewed as running concurrently, and the tools processing litmus tests carry out something similar to a full state-space search. These tools then classify all valid executions based on whether the logic expression in the `exists` clause on lines 26-27 holds or not. Please note that this expression is evaluated only "at the end of time" for each valid execution, "after all the dust has settled". For example, if the

[1] Back in ancient times, undeclared FORTRAN variables beginning with the letters "i", "j", "k", "l", "m", or "n" were implicitly `INTEGER`, and all other undeclared variables were implicitly `REAL`. Modern best practices dictate that a `IMPLICIT NONE` declaration be used, which requires that all variables be explicitly declared, thus preventing any number of hard-to-find bugs stemming from variable-name typos.

Listing 3.1: Two-Process Load Buffering

```
C LB+o-sr-o+rlk-o-o-rulk
{
}

P0(atomic_int *x0, atomic_int *x1)
{
  int r1;

  r1 = atomic_load_explicit(x0, memory_order_relaxed);
  synchronize_rcu();    /* std::synchronize_rcu(); */
  atomic_store_explicit(x1, 1, memory_order_relaxed);
}


P1(atomic_int *x0, atomic_int *x1)
{
  int r1;

  rcu_read_lock();      /* std::rcu_reader rr; */
  r1 = atomic_load_explicit(x1, memory_order_relaxed);
  atomic_store_explicit(x0, 1, memory_order_relaxed);
  rcu_read_unlock();
}

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

expression evaluates to `true` midway through a particular valid execution, but evaluates to `false` at the end of that execution, then that execution will be classified as resulting in a `false` outcome.

Note also that the `/\` in the litmus test denotes logical AND. The individual terms are evaluated in a manner similar to the initialization statements, so that `1:r1=y0` evaluates to `true` iff `P1()`'s local variable `r1` ends up containing a value equal to the address of global variable `y0`. Similarly, `1:r2=1` evaluates to `true` iff `P1()`'s local variable `r2` ends up containing the value 1.

The following section will discuss litmus tests that are less frequently used in practice, but which more clearly illustrate ordering requirements.

3 Litmus Tests

Section 3.1 presents a litmus test illustrating RCU's fundamental grace-period guarantee and Section 3.2 presents more ornate litmus tests. In the Linux kernel, any RCU implementation providing the fundamental grace-period guarantee can be proven to also satisfy the guarantees illustrated by the more ornate litmus tests. Proving (or disproving) this result in the context of the C++ memory model is important future work.

3.1 Basic Litmus Test

Listing 3.1 contains a basic litmus test that illustrates the lowest-level ordering guarantees that RCU provides. This idiom is used in change-of-state use cases where the state change need not be instantaneously visible throughout the application, but where specific processing must wait until the change becomes globally visible. In other words, if `P1()` sees `P0()`'s write, `P0()` must be guaranteed not to see `P1()`'s write, and vice versa. Similar guarantees must be provided for store buffering and message-passing litmus tests, but for simplicity, this paper focuses on load buffering.

The ordering guarantee has three cases:

1. P0()'s r1 ends with the value 1.
2. P1()'s r1 ends with the value 1.
3. Both r1 variables end with the value 0.

For the first case, if P0()'s r1 ends with the value 1, some part of P1()'s RCU read-side critical section (spanning lines 19-22) precedes the call to P0's synchronize_rcu(). RCU therefore requires that the destructor for P1()'s RCU read-side critical section must in some sense precede the return from P0's synchronize_rcu(), whether "precedes" means synchronizes with, happens before, strongly happens before, or something else. Either way, it is necessary that anything within or before P1()'s RCU read-side critical section must happen before everything sequenced after P0's synchronize_rcu(). Therefore, when P0()'s r1 ends with the value 1, then P1()'s r1 must end with the value 0, so that the exists clause's expression cannot evaluate to true.

For the second case, if P1()'s r1 ends with the value 1, some part of P1()'s RCU read-side critical section (spanning lines 19-22) follows the return from P0's synchronize_rcu(). RCU therefore requires that the call to P0's synchronize_rcu() must in some sense precede the constructor for P1()'s RCU read-side critical section, whether "precedes" means synchronizes with, happens before, strongly happens before, or something else. Either way, it is necessary that anything sequenced before P0's synchronize_rcu() must happen before everything within or after P1()'s RCU read-side critical section. Therefore, when P1()'s r1 ends with the value 1, then P0()'s r1 must end with the value 0, so that the exists clause's expression cannot evaluate to true.

For the third case, we have no ordering information. It is still the case that there is ordering because RCU read-side critical sections are not allowed to span synchronize_rcu() invocations. So it is the case that either:

1. Anything within or before P1()'s RCU read-side critical section happens before everything sequenced after P0's synchronize_rcu(), or
2. Anything sequenced before P0's synchronize_rcu() must happen before everything within or after P1()'s RCU read-side critical section.

However, it is not possible to distinguish between these two outcomes.

3.2 Ornate Litmus Tests

Section 3.2.1 presents three-process load buffering with one reader, Section 3.2.2 presents three-process load buffering with two readers, Section 3.2.3 presents four-process load buffering, and finally Section 3.2.4 presents six-process load buffering.

3.2.1 Three-Process Load Buffering, One Reader

Listing 3.2 shows a litmus test with three processes, two invoking synchronize_rcu() and a third containing an RCU read-side critical section.

The interactions between P1() and P2() on the one hand and between P2() and P0() on the other can be modeled in a manner similar to the interactions between P0() and P1() in Listing 3.1. In particular, if P0()'s and P2()'s r1 both obtain the value 1, then:

Listing 3.2: Three-Process Load Buffering, One Reader

```
C LB+o-sr-o+o-sr-o+rlk-o-o-rulk
{
}

P0(atomic_int *x0, atomic_int *x1)
{
  int r1;

  r1 = atomic_load_explicit(x0, memory_order_relaxed);
  synchronize_rcu();    /* std::synchronize_rcu(); */
  atomic_store_explicit(x1, 1, memory_order_relaxed);
}


P1(atomic_int *x1, atomic_int *x2)
{
  int r1;

  r1 = atomic_load_explicit(x1, memory_order_relaxed);
  synchronize_rcu();    /* std::synchronize_rcu(); */
  atomic_store_explicit(x2, 1, memory_order_relaxed);
}


P2(atomic_int *x2, atomic_int *x0)
{
  int r1;

  rcu_read_lock();      /* std::rcu_reader rr; */
  r1 = atomic_load_explicit(x2, memory_order_relaxed);
  atomic_store_explicit(x0, 1, memory_order_relaxed);
  rcu_read_unlock();
}

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

1. Anything within or before `P2()`'s RCU read-side critical section happens before everything sequenced after `P0`'s `synchronize_rcu()`, and

2. Anything sequenced before `P1`'s `synchronize_rcu()` must happen before everything within or after `P2()`'s RCU read-side critical section.

However, careful consideration of RCU's safety properties indicates that this situation requires everything preceding `P1()`'s `synchronize_rcu()` to happen before everything following `P0()`'s `synchronize_rcu()`, so that `P1()`'s `r1` must obtain the value 0. In other words, the choice of ordering type from `synchronize_rcu()` to a subsequent `rcu_reader`'s constructor and the choice of ordering from a `rcu_reader`'s destructor to a subsequent `synchronize_rcu()` must allow some sort of transitivity.

Alternatively, suppose that `P1()`'s and `P2()`'s `r1` obtain the value 1. In this case, everything preceding both `P0()`'s and `P1()`'s invocations of `synchronize_rcu()` must happen before everything within or after `P2()`'s RCU read-side critical section, so that `P0()`'s `r1` would obtain the value 0. Furthermore, `synchronize_rcu()` has all the ordering semantics of the `atomic_memory_fence(memory_order_seq_cst)` full fence, which means that line 10 synchronizes with line 20, which implies that line 9 happens before line 21.

Finally, suppose that `P0()`'s `r1` and `P1()`'s `r1` both obtain the value 1. In this case, everything within or before `P2()`'s RCU read-side critical section must happen before everything following either `P0()`'s and `P1()`'s invocations of `synchronize_rcu()`, so that `P2()`'s `r1` would obtain the value 0. Again, `synchronize_rcu()` full-fence semantics imply that line 9 happens before line 21.

3.2.2 Three-Process Load Buffering, Two Readers

Listing 3.3 shows two reader processes (`P1()` and `P2()`) and a third process with not one but two invocations of `synchronize_rcu()`.

Suppose that both `P0()`'s and `P1()`'s `r1` both obtain the value 1. Then everything within or before `P2()`'s RCU read-side critical section, including line 32, happens before `P0()`'s first `synchronize_rcu()` returns. Because `P1()`'s `r1` obtains the value 1, the call to `P0()`'s first invocation of `synchronize_rcu()` happens before everything within or after `P1()`'s RCU read-side critical section, including line 22. This means that line 32 happens before line 22, and therefore that `P2()`'s `r1` must obtain the value 0.

Suppose that both `P0()`'s and `P2()`'s `r1` both obtain the value 1. Because `P0()`'s `r1` obtains 1, everything within or before `P2()`'s RCU read-side critical section happens before anything following the return from `P0()`'s first invocation of `synchronize_rcu()`, a set that includes the entirety of `P0()`'s second invocation of `synchronize_rcu()`. Now, `P1()`'s write to `x2` in some sense precedes `P2()`'s read because `P2()`'s `r1` obtained the value 1,[2] which in turn means that `P1()`'s write to `x2` in some sense also precedes `P0()`'s second invocation of `synchronize_rcu()`. Therefore, everything within or before `P1()`'s RCU read-side critical section must happen before the return from `P0()`'s second `synchronize_rcu()`, which means that `P2()`'s `r1` must obtain the value zero.

Finally, suppose that both `P1()`'s and `P2()`'s `r1` both obtain the value 1. Because `P1()`'s `r1` obtains 1, anything preceding the call to `P0()`'s second `synchronize_rcu()` invocation (including `P0()`'s first invocation of `synchronize_rcu()`) must happen

[2] Recall that RCU read-side critical sections are in no way shape or form allowed to completely overlap the execution of any `synchronize_rcu()` invocation.

Listing 3.3: Three-Process Load Buffering, Two Readers

```
C LB+o-sr-sr-o+rlk-o-o-rulk+rlk-o-o-rulk
{
}

P0(atomic_int *x0, atomic_int *x1)
{
  int r1;
  int r2;

  r1 = atomic_load_explicit(x0, memory_order_relaxed);
  synchronize_rcu();    /* std::synchronize_rcu(); */
  synchronize_rcu();    /* std::synchronize_rcu(); */
  atomic_store_explicit(x1, 1, memory_order_relaxed);
}


P1(atomic_int *x1, atomic_int *x2)
{
  int r1;

  rcu_read_lock();      /* std::rcu_reader rr; */
  r1 = atomic_load_explicit(x1, memory_order_relaxed);
  atomic_store_explicit(x2, 1, memory_order_relaxed);
  rcu_read_unlock();
}


P2(atomic_int *x2, atomic_int *x0)
{
  int r1;

  rcu_read_lock();      /* std::rcu_reader rr; */
  r1 = atomic_load_explicit(x2, memory_order_relaxed);
  atomic_store_explicit(x0, 1, memory_order_relaxed);
  rcu_read_unlock();
}

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

Listing 3.4: Four-Process Load Buffering

```
C LB+o-sr-o+o-sr-o+rlk-o-o-rulk+rlk-o-o-rulk
{
}

P0(atomic_int *x0, atomic_int *x1)
{
  int r1;

  r1 = atomic_load_explicit(x0, memory_order_relaxed);
  synchronize_rcu();    /* std::synchronize_rcu(); */
  atomic_store_explicit(x1, 1, memory_order_relaxed);
}


P1(atomic_int *x1, atomic_int *x2)
{
  int r1;

  r1 = atomic_load_explicit(x1, memory_order_relaxed);
  synchronize_rcu();    /* std::synchronize_rcu(); */
  atomic_store_explicit(x2, 1, memory_order_relaxed);
}


P2(atomic_int *x2, atomic_int *x3)
{
  int r1;

  rcu_read_lock();      /* std::rcu_reader rr; */
  r1 = atomic_load_explicit(x2, memory_order_relaxed);
  atomic_store_explicit(x3, 1, memory_order_relaxed);
  rcu_read_unlock();
}


P3(atomic_int *x0, atomic_int *x3)
{
  int r1;

  rcu_read_lock();      /* std::rcu_reader rr; */
  r1 = atomic_load_explicit(x3, memory_order_relaxed);
  atomic_store_explicit(x0, 1, memory_order_relaxed);
  rcu_read_unlock();
}

exists
(0:r1=1 /\ 1:r1=1 /\ 2:r1=1 /\ 3:r1=1)
```

before anything within or after `P1()`'s RCU read-side critical section. Again, `P1()`'s write to `x2` in some sense precedes `P2()`'s read because `P2()`'s `r1` obtained the value 1, which in turn means that `P2()`'s read from `x2` in some sense also follows `P0()`'s first invocation of `synchronize_rcu()`. Therefore, everything preceding the call to `P0()`'s first invocation of `synchronize_rcu()` must happen before everything within or after `P2()`'s RCU read-side critical section, which means that `P0()`'s `r1` must obtain the value zero.

This example shows that a consecutive pair of `synchronize_rcu()` invocations is stronger than that of a single invocation: If `P0()` had only one `synchronize_rcu()`, the resulting litmus test would be allowed, that is, all three instances of `r1` could obtain the value 1. In contrast, a consecutive pair of `atomic_thread_fence(memory_order_seq_cst)` invocations is no stronger than a single invocation.

3.2.3 Four-Process Load Buffering

Listing 3.4 shows a four-process litmus test two processes using `synchronize_rcu()` and two processes having RCU read-side critical sections. The two interesting orders occur when `P0()`'s, `P1()`'s, and `P3()`'s `r1` all obtain the value 1 and when `P1()`'s, `P2()`'s, and `P3()`'s `r1` all obtain the value 1. In both orderings, `P1()`'s `r1` obtains the value 1, which means that the two invocations of `synchronize_rcu()` are serialized, that is, `P0()`'s `synchronize_rcu()` ends before `P1()`'s `synchronize_rcu()` begins.

In the first order, where `P0()`'s, `P1()`'s, and `P3()`'s `r1` all obtain the value 1, because `P0()`'s `r1` obtains the value 1, everything within or preceding `P3()`'s RCU read-side critical section happens before `P0()`'s invocation of `synchronize_rcu()`, in particular, line 41 happens before line 11. Because `P3()`'s `r1` obtains the value 1, line 31 happens before line 11. Now, `P2()`'s RCU read-side critical section is in no way, shape, or form allowed to completely overlap `P1()`'s invocation of `synchronize_rcu()`, which means that everything within or preceding `P2()`'s RCU read-side critical section happens before everything following `P1()`'s invocation of `synchronize_rcu()`, in particular, line 30 happens before line 21. This means that `P2()`'s `r1` must obtain the value zero.

In the second order, when `P1()`'s, `P2()`'s, and `P3()`'s `r1` all obtain the value 1, because `P2()`'s obtains the value 1, everything preceding `P1()`'s invocation of `synchronize_rcu()` happens before everything within and after `P2()`'s RCU read-side critical section, in particular, line 19 happens before line 31. Because `P3()`'s `r1` obtains the value 1, line 19 happens before line 41. Now, `P3()`'s RCU read-side critical section is in no way, shape, or form allowed to completely overlap `P0()`'s invocation of `synchronize_rcu()`, which means that everything preceding `P0()`'s invocation of `synchronize_rcu()` happens before everything within or after `P3()`'s RCU read-side critical section, in particular, line 9 happens before line 42. This means that `P0()`'s `r1` must obtain the value zero.

3.2.4 Six-Process Load Buffering

Listing 3.5 shows a six-process litmus test with three processes using `synchronize_rcu()` and three processes having RCU read-side critical sections. The two interesting orders occur when `P0()`'s, `P1()`'s, `P2()`'s, `P4()`'s, and `P5()`'s `r1` all obtain the value 1 and when `P1()`'s, `P2()`'s, `P3()`, `P4()`, and `P5()`'s `r1` all obtain the value 1. In both orderings, `P1()`'s and `P2()`'s `r1` obtain the value 1, which means that the three invocations of `synchronize_rcu()` are serialized, that is, `P0()`'s `synchronize_rcu()` ends before `P1()`'s `synchronize_rcu()` begins and `P1()`'s `synchronize_rcu()` ends before `P2()`'s `synchronize_rcu()` begins.

In the first order, where `P0()`'s, `P1()`'s, `P2()`'s, `P4()`'s, and `P5()`'s `r1` all obtain the value 1, because `P0()`'s `r1` obtains the value 1, everything within or preceding `P5()`'s RCU read-side critical section happens before `P1()`'s invocation of `synchronize_rcu()`, in particular, line 62 happens before line 11. Because `P5()`'s `r1` obtains the value 1, line 52 also in some sense precedes line 11. Now, `P4()`'s RCU read-side critical section is in no way, shape, or form allowed to completely overlap `P1()`'s invocation of `synchronize_rcu()`, which means that everything within or preceding `P4()`'s RCU read-side critical section happens before everything following `P1()`'s invocation of `synchronize_rcu()`. In particular, line 51 happens before line 21. Because `P4()`'s `r1` obtains the value 1, line 41 also in some way precedes line 21. But `P3()`'s RCU read-side critical section is in no way, shape, or form allowed

Listing 3.5: Six-Process Load Buffering

```
C LB+o-sr-o+o-sr-o+o-sr-o+rlk-o-o-rulk+rlk-o-o-rulk+rlk-o-o-rulk
{
}

P0(atomic_int *x0, atomic_int *x1)
{
  int r1;

  r1 = atomic_load_explicit(x0, memory_order_relaxed);
  synchronize_rcu();    /* std::synchronize_rcu(); */
  atomic_store_explicit(x1, 1, memory_order_relaxed);
}


P1(atomic_int *x1, atomic_int *x2)
{
  int r1;

  r1 = atomic_load_explicit(x1, memory_order_relaxed);
  synchronize_rcu();    /* std::synchronize_rcu(); */
  atomic_store_explicit(x2, 1, memory_order_relaxed);
}


P2(atomic_int *x2, atomic_int *x3)
{
  int r1;

  r1 = atomic_load_explicit(x2, memory_order_relaxed);
  synchronize_rcu();    /* std::synchronize_rcu(); */
  atomic_store_explicit(x3, 1, memory_order_relaxed);
}


P3(atomic_int *x3, atomic_int *x4)
{
  int r1;

  rcu_read_lock();      /* std::rcu_reader rr; */
  r1 = atomic_load_explicit(x3, memory_order_relaxed);
  atomic_store_explicit(x4, 1, memory_order_relaxed);
  rcu_read_unlock();
}


P4(atomic_int *x4, atomic_int *x5)
{
  int r1;

  rcu_read_lock();      /* std::rcu_reader rr; */
  r1 = atomic_load_explicit(x4, memory_order_relaxed);
  atomic_store_explicit(x5, 1, memory_order_relaxed);
  rcu_read_unlock();
}


P5(atomic_int *x0, atomic_int *x5)
{
  int r1;

  rcu_read_lock();      /* std::rcu_reader rr; */
  r1 = atomic_load_explicit(x5, memory_order_relaxed);
  atomic_store_explicit(x0, 1, memory_order_relaxed);
  rcu_read_unlock();
}

exists
(0:r1=1 /\ 1:r1=1 /\ 2:r1=1 /\ 3:r1=1 /\ 4:r1=1 /\ 5:r1=1)
```

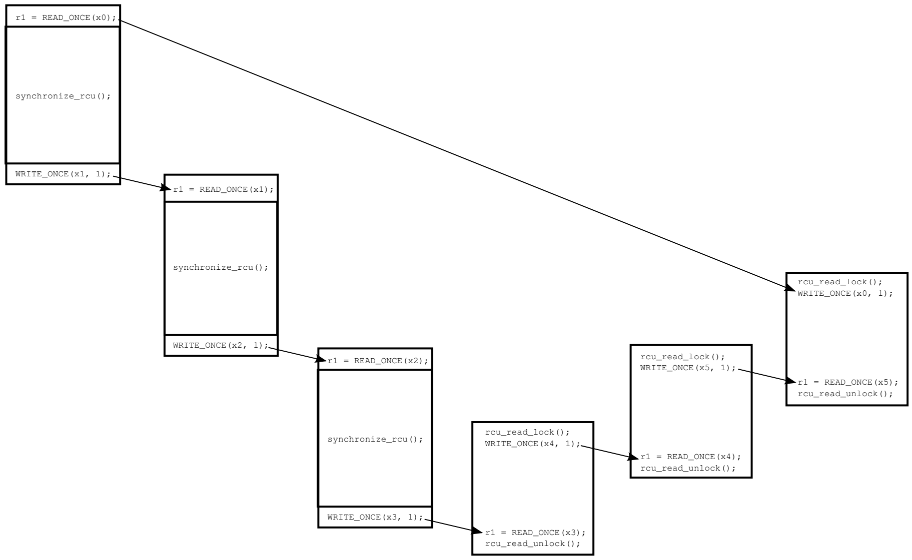

Figure 1: Visual Representation of Six-Process Load Buffering

to completely overlap `P2()`'s invocation of `synchronize_rcu()`, which means that everything within or preceding `P3()`'s RCU read-side critical section happens before everything following `P2()`'s invocation of `synchronize_rcu()`. In particular, line 40 happens before line 31. This means that `P3()`'s `r1` must obtain the value zero.

In the second order, when `P1()`'s, `P2()`'s, `P3()`'s, `P4()`'s, and `P5()`'s `r1` all obtain the value 1, because `P3()`'s obtains the value 1, everything preceding `P2()`'s invocation of `synchronize_rcu()` happens before everything within and after `P3()`'s RCU read-side critical section. Because `P4()`'s `r1` obtains the value 1, line 29 happens before line 51. Now, `P4()`'s RCU read-side critical section is in no way, shape, or form allowed to completely overlap `P1()`'s invocation of `synchronize_rcu()`, which means that everything preceding `P1()`'s invocation of `synchronize_rcu()` happens before everything within or after `P4()`'s RCU read-side critical section, in particular, line 19 happens before line 52. Because `P5()`'s `r1` obtains the value 1, line 19 happens before line 63. Now, `P5()`'s RCU read-side critical section is in no way, shape, or form allowed to completely overlap `P0()`'s invocation of `synchronize_rcu()`, which means that everything preceding `P0()`'s invocation of `synchronize_rcu()` happens before everything within or after `P5()`'s RCU read-side critical section, in particular, line 11 happens before line 63. This means that `P0()`'s `r1` must obtain the value zero.

4 RCU Ordering Rationale

To see why `synchronize_rcu()` includes the ordering semantics of a full memory fence, consider `P0()` and `P1()` in Listing 3.4. Without full-fence semantics, it might be that `P1()`'s `r1` would obtain the value 1, but line 9's read not happen before line 21's write. This outcome would be quite surprising to most users, and furthermore it would be quite difficult to create a `synchronize_rcu()` implementation that interacted correctly with RCU read-side critical sections, but that failed to provide full-fence ordering semantics.

Another way to estimate ordering effects in simple RCU litmus tests is to assume that RCU grace periods are a given fixed duration and that RCU read-side critical sections are a slightly shorter fixed duration. Then if maximally obtuse memory-access reorderings are applied, the RCU relationships may be easily diagrammed. For example, the relationships for Listing 3.5, assuming that all `r1` local variables other than that of `P0()` obtain the value one, are shown in Figure 1. Further consideration of this estimation approach gives rise to the *counting rule* for pure RCU litmus tests: As long as there are at least as many RCU grace periods as there are RCU read-side critical sections in the litmus test's cycle, the outcome will be forbidden.

This leads to the question that is the whole point of this paper: What ordering is required between `rcu_reader` constructors and destructors on the one hand and `synchronize_rcu()` on the other?

5 Candidate Solutions

The following sections propose various solutions to the RCU ordering problem. Note that these proposals are not necessarily all mutually exclusive.

5.1 Synchronizes-With

This section documents the initial state of D0556R4 ("Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU)").

This proposal assumes that `synchronize_rcu()` is implemented using a call to `rcu_retire()` whose deleter[3] awakens the thread that invoked `rcu_retire()`, which allows specification of ordering to focus on `rcu_retire()`, and, by extension, the `retire()` member function of `rcu_obj_base`.

This proposal guarantees that for each instance R of `rcu_reader`, one of the following two things hold:

1. `rcu_retire` *synchronizes with* R's constructor, or
2. R's destructor *synchronizes with* the invocation of the deleter.

5.2 Happens-Before and Synchronizes-With

This section documents Andrew Hunter's proposed update to D0556R4 ("Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU)").

This proposal also assumes that `synchronize_rcu()` is implemented using a call to `rcu_retire()` whose deleter awakens the thread that invoked `rcu_retire()`, which again allows specification of ordering to focus on `rcu_retire()`, and, by extension, the `retire()` member function of `rcu_obj_base`.

This proposal guarantees that for each instance R of `rcu_reader`, one of the following two things hold:

1. `rcu_retire` *happens before* R's constructor, or
2. R's destructor *synchronizes with* the invocation of the deleter.

[3] What the Linux kernel calls an RCU callback, C++ calls a deleter.

Listing 5.1: Split `synchronize_rcu()`

```
synchronize_rcu()
{
  atomic_thread_fence(memory_order_seq_cst);
  stmt1: /* nop */ ;
  stmt2: /* nop */ ;
  atomic_thread_fence(memory_order_seq_cst);
}
```

In other words, this proposal is the same as that of Section 5.1, except that the first option's *synchronizes with* has become *happens before*.

5.3 Split `synchronize_rcu()`

The preceding sections assume that `synchronize_rcu()` is implemented as an `rcu_retire()` whose callback awakens the thread that invoked the `rcu_retire()`. This section instead handles `synchronize_rcu()` directly, as proposed by Alan Stern.

This proposal splits `synchronize_rcu()` as shown in Listing 5.1. Given this split, the RCU ordering requirements could be expressed as: For any read-side critical section and any call to `synchronize_rcu()` in different threads, the behavior should be as if either:

1. The `rcu_reader` destructor *synchronizes with* `stmt2` (this is the case where the corresponding RCU read-side critical section comes before the end of the grace period), or

2. The `stmt1` in `synchronize_rcu()` *synchronizes with* the `rcu_reader` constructor (this is the case where the start of the RCU grace period comes before the corresponding RCU read-side critical section).

Note that the above definition best matches Linux-kernel RCU semantics given a fully functional strong fence. This proposal therefore assumes that one of the proposals for strengthening C++'s `atomic_thread_fence(memory_order_seq_cst)` eventually becomes part of the standard.

Within a given thread, for any read-side critical section and any call to `synchronize_rcu()`:

1. The `rcu_reader` destructor is *sequenced before* `synchronize_rcu()`, or

2. The invocation of `synchronize_rcu()` is *sequenced before* the `rcu_reader` constructor.

Placing a `synchronize_rcu()` between a `rcu_reader`'s constructor and destructor is not a strategy to win. If you are lucky, all that will happen is a deadlock. If you are not so lucky, you will invoke undefined behavior.

6 Prototype C11 Memory Model Including RCU

This section presents a C11 memory model that includes RCU. As such, this model replaces `rcu_reader` constructors with `rcu_read_lock()` and `rcu_reader` destructors with `rcu_read_unlock()`. This is of course merely a syntactic mapping that does not affect the underlying memory-ordering semantics.

Listing 6.1: Top-level `c11.cat` Model

```
1 "C++11" withinit
2
3 include "c11_base.cat"
4
5 include "c11-sg1-from-olivier-via-paul.cat"
6
7 include "c11-sc-fences-vafeiadis-et-al17.cat"
8
9 include "rcu-utils.cat"
10
11 procedure rcu-c11(gp,rscs) =
12   let A-cumul = scb?; rfe?
13   let B-cumul = hb*
14   let strong-fence = fencerel(F & SC)
15   call rcu(A-cumul, B-cumul, strong-fence, gp, rscs)
16 end
17
18 call rcu-c11(gp,rscs)
```

This model may be run on a litmus test (for example, `C11-RCU-MP.litmus`) using the command "`herd7 -c11 -cat c11.cat C11-RCU-MP.litmus`". This section describes the RCU-related portions of the `cat` files, but please treat these descriptions with extreme scepticism, as they have not yet received any review. Any errors are the sole property of Paul E. McKenney.

Listing 6.1 shows a top-level `cat` model of C11 including RCU. Line 3 includes a file containing the base C11 model, line 5 includes a file containing the modifications to C11 proposed by Olivier Giroux [3] line 7 includes a file containing the modifications proposed by Vafeiadis et al. [4], and finally line 9 includes a file containing experimental definitions for RCU.

Lines 11–16 define the `rcu-c11` procedure, which allows easy specification of upstream ordering (line 12), outbound ordering (line 13), and fence strength (line 14). These specifications are passed to the `rcu` procedure, which is defined in `rcu-utils.cat`. Line 18 invokes the `rcu-c11` procedure.

Listings 6.2 and 6.2 display the base C11 model found in the `herdtools` distribution. Listings 6.4 and 6.5 display Olivier's and Vafeiadis et al.'s proposed modifications, respectively. These are shown for the benefit of readers wishing to see the full model, but are not described as they are not directly relevant to RCU.

The proposed RCU memory model is shown in Listing 6.6. This model was produced by Jade and Luc, and was based on the RCU portion of the Linux-kernel memory model [1, 2]. Lines 4–20 extract a relation linking the beginnings and ends of outermost RCU read-side critical sections into the variable `crit`. This extraction process starts with the pre-defined set variables `Rcu-lock` (which is a set of all `rcu_read_lock()` statements in the litmus test) and `Rcu_unlock` (which is a set of all `rcu_read_unlock()` statements in the litmus test).

The first step of this extraction uses six mutually recursive set functions on lines 4-13. These functions are as follows:

`unmatched-locks`: This set contains `rcu_read_lock()` statements whose matching `rcu_read_unlock()` either does not exist or has not yet been located. This relation is computed by removing all matched `rcu_read_lock()` statements from the `Rcu-lock` set.

`unmatched-unlocks`: This set contains `rcu_read_unlock()` statements whose matching `rcu_read_lock()` either does not exist or has not yet been located. This

Listing 6.2: Base C11 Memory Model, Part 1 of 2

```
"C++11" withinit

include "c11_cos.cat"
include "c11_los.cat"


let asw = I * (M \ I)
let sb = po
let mo = co

let cacq = ACQ | (SC & (R | F)) | ACQ_REL | (F & CON)
let crel = REL | (SC & (W | F)) | ACQ_REL
let ccon = R & CON

let fr = rf^-1 ; mo

let dd = (data | addr)+

let fsb = sb & (F * _)
let sbf = sb & (_ * F)

(* release_acquire_fenced_synchronizes_with,
   hypothetical_release_sequence_set,
   release_sequence_set *)

(* OLD: let rs = [crel] ; fsb? ; [A & W] ;
   (((mo ; [rmw]) | coi) & ~(coe ; [!rmw] ; mo))? *)

let rs_prime = int | (_ * (R & W))
let rs = mo & rs_prime \ ((mo \ rs_prime) ; mo)

(* OLD: let swra = ext (rs ; rf ; [A] ; sbf? ; [cacq]) *)
let swra =
 ext &
 (toid(crel) ; fsb? ; toid(A & W) ; rs? ; rf ;
 toid(R & A) ; sbf? ; toid(cacq))

let swul = ext & (toid(UL) ; lo ; toid(LK))
let pp_asw = asw \ (asw ; sb)
let sw = pp_asw | swul | swra

(* with_consume_cad_set,
   dependency_ordered_before *)
let cad = ((rf & sb) | dd)+
let dob =
 (ext &
  (toid(W & crel) ; fsb? ; toid(A & W) ;
   rs?; rf; toid(ccon)));
 cad?

```

Listing 6.3: Base C11 Memory Model, Part 2 of 2

```
(* happens_before,
   inter_thread_happens_before,
   consistent_hb *)
let ithbr = sw | dob | (sw ; sb)
let ithb = (ithbr | (sb ; ithbr))+
let hb = sb | ithb
acyclic hb as Hb

show (hb \ (IW * _)) & ext as hb

(* coherent_memory_use *)
let hbl = hb & loc

irreflexive ((rf^-1)? ; mo ; rf? ; hb) as Coh

(* visible_side_effect_set *)
let vis = (hbl & (W * R)) \ (hbl; toid(W) ; hbl)

(* consistent_atomic_rf *)
irreflexive (rf ; hb) as Rf

(* consistent_non_atomic_rf  *)
empty ((rf ; [R\A]) \ vis) as NaRf
empty [FW\A];hbl;[W] as NaRf (* implicit read of Na final writes.. *)

irreflexive (rf | (mo ; mo ; rf^-1) | (mo ; rf)) as Rmw


(* locks_only_consistent_lo *)
irreflexive (lo ; hb) as Lo1

(* locks_only_consistent_locks *)
irreflexive
   (toid(LS) ; lo^-1 ; toid(LS) ; ~(lo ; toid(UL) ; lo)) as Lo2


(* data_races *)
let Mutex = UL|LS
let cnf = (((W * _) | (_ * W)) & loc) \ ((Mutex * _) | (_ * Mutex))
let dr = ext & (cnf \ hb \ (hb^-1) \ (A * A))

(* unsequenced_races *)
let ur = int & ((W * M) | (M * W)) &
         loc & ~id & ~(sb+) & ~((sb+)^-1)

(* locks_only_good_mutex_use,  locks_only_bad_mutexes *)

let bl = (toid(LS); (sb & lo); toid(LK)) & ~(lo; toid(UL); lo)

let losbwoul = (sb & lo & ~(lo; toid(UL); lo))

let lu = toid(UL) &
  ~(toid(UL) ; losbwoul^-1 ; toid(LS) ; losbwoul ; toid(UL))
```

Listing 6.4: Olivier's Proposed C11 Modifications

```
"C++11 SG1 proposal for SC atomics from Olivier via Paul" withinit

include "c11_base.cat"

let r1 = (po | (sw & (SC * SC)) | po;hb;po)+ (*included in hb, but not all of hb*)
let r2 = fsb? ; mo ; sbf?
let r3 = rf^-1; toid(SC) ; mo
let r4 = rf^-1 ; hbl ; toid(W)
let r5 = fsb ; fr
let r6 = fr ; sbf
let r7 = fsb ; fr ; sbf

let scp = r1|r2|r3|r4|r5|r6|r7

acyclic (((SC * SC) & scp) \ id) as Spartial
show scp


undefined_unless empty dr as Dr
undefined_unless empty ur as unsequencedRace
undefined_unless empty bl as badLock
undefined_unless empty lu as badUnlock
```

Listing 6.5: Vafeiadis et al.'s Proposed C11 Modifications

```
"C++11 SG1 proposal for SC atomics from Olivier via Paul" withinit

include "c11_base.cat"

let r1 = (po | (sw & (SC * SC)) | po;hb;po)+ (*included in hb, but not all of hb*)
let r2 = fsb? ; mo ; sbf?
let r3 = rf^-1; toid(SC) ; mo
let r4 = rf^-1 ; hbl ; toid(W)
let r5 = fsb ; fr
let r6 = fr ; sbf
let r7 = fsb ; fr ; sbf

let scp = r1|r2|r3|r4|r5|r6|r7

acyclic (((SC * SC) & scp) \ id) as Spartial
show scp


undefined_unless empty dr as Dr
undefined_unless empty ur as unsequencedRace
undefined_unless empty bl as badLock
undefined_unless empty lu as badUnlock
```

Listing 6.6: Proposed C11 RCU Memory Model

```
"RCU utils"

(* Compute matching pairs of nested Rcu-lock and Rcu-unlock *)
let matched = let rec
      unmatched-locks = Rcu-lock \ domain(matched)
  and unmatched-unlocks = Rcu-unlock \ range(matched)
  and unmatched = unmatched-locks | unmatched-unlocks
  and unmatched-po = (unmatched * unmatched) & po
  and unmatched-locks-to-unlocks = (unmatched-locks *
      unmatched-unlocks) & po
  and matched = matched | (unmatched-locks-to-unlocks \
    (unmatched-po ; unmatched-po))
  in matched

(* Validate nesting *)
flag ~empty Rcu-lock \ domain(matched) as unbalanced-rcu-locking
flag ~empty Rcu-unlock \ range(matched) as unbalanced-rcu-locking

(* Outermost level of nesting only *)
let crit = matched \ (po^-1 ; matched ; po^-1)

let rscs = po; crit^-1; po

let gp = (po & ( _ * Sync-rcu));po?

let pb(A,B,sf) = A; sf; B
let link(A,B,sf) = B; (pb(A,B,sf))*; A

procedure rcu(A,B,sf,gp,rscs) =
  let mylink = link(A,B,sf)
  let gp-link = gp ; mylink
  let rscs-link = rscs ; mylink
  let rec rcu-path = gp-link |
                     (gp-link ; rscs-link) |
                     (rscs-link ; gp-link) |
                     (rcu-path ; rcu-path) |
                     (gp-link ; rcu-path ; rscs-link) |
                     (rscs-link ; rcu-path ; gp-link)
  irreflexive rcu-path as rcu

  (*flag: mylink; strong-fence; mylink included in mylink
         if flag raised then we got it wrong
         this is essentially Lemma 1*)
  flag ~empty((mylink; sf; mylink) \ mylink) as rcu-fail
end
```

relation is computed by removing all matched `rcu_read_unlock()` statements from the `Rcu-unlock` set.

`unmatched:` This set is the union of `unmatched-locks` and `unmatched-unlocks` called out above, that is, the set of `rcu_read_lock()` and `rcu_read_unlock()` statements whose matching statement either does not exist or has not yet been located.

`unmatched-po:` This relation links any given unmatched `rcu_read_lock()` statement or `rcu_read_unlock()` statement to all subsequent unmatched statements within the same process. This relation is computed by forming the cross product of `unmatched` with itself, then restricting the resulting relation to a given process by intersecting with the program-order relation `po`.

`unmatched-locks-to-unlocks:` This relation links `rcu_read_lock()` statement that have not yet been matched to all subsequent unmatched `rcu_read_unlock()` statements within the same process. This relation is computed by forming the cross product of `unmatched_locks` with `unmatched-unlocks`, then restricting the resulting relation to a given process by intersecting with the program-order relation `po`.

`matched:` This relation links each `rcu_read_lock()` statement with the matching `rcu_read_unlock()` statement. If there is misnesting, and thus unmatched `rcu_read_lock()` or `rcu_read_unlock()` statements, then these unmatched statements will not appear in the `matched` relation. This relation is computed by removing all pairs of `rcu_read_lock()` and `rcu_read_unlock()` statements from `unmatched-locks-to-unlocks` that have at least one unmatched `rcu_read_lock()` or `rcu_read_unlock()` statement between them. Each step of recursion therefore adds innermost unmatched pairs of `rcu_read_lock()` and `rcu_read_unlock()` statements to the `matched` relation.

Lines 16 and 17 can then flag misnesting. Line 16 checks for elements of the `Rcu-lock` set that do not appear in the domain of the `matched` relation, that is, for unmatched `rcu_read_lock()` statements. Similarly, line 17 checks for elements of the `Rcu-unlock` set that do not appear in the range of the `matched` relation, that is, for unmatched `rcu_read_unlock()` statements.

Finally, line 20 forms the relation `crit`, which contains outermost matched pairs of `rcu_read_lock()` and `rcu_read_unlock()` statements. The trick is that `po^-1 ; matched ; po^-1` matches any pair of `rcu_read_lock()` and `rcu_read_unlock()` statements that are within an enclosing pair of `rcu_read_lock()` and `rcu_read_unlock()` statements. Removing this relation from the `matched` relation results in a relation having only outermost matched pairs of `rcu_read_lock()` and `rcu_read_unlock()` statements.

The remainder of Listing 6.6 (lines 24–45) defines the relationship between RCU read-side critical sections and RCU grace periods.

Line 24 defines the gp grace-period relationship, which any statement preceding a `synchronize_rcu()` statement to that `synchronize_rcu()` statement and to any statement following that `synchronize_rcu()` statement. This differs from the relations used for memory fences, which do not link to the memory fence itself. The reason for this difference is that, unlike memory fences, repeated `synchronize_rcu()` statements have stronger semantics than do isolated `synchronize_rcu()` statements.

Lines 26 and 27 define the relation between successive RCU read-side critical sections and/or RCU grace periods. Recall that the arguments to these functions are defined by the `rcu_c11()` procedure back on lines 11–16 of Listing 6.1. Back on Listing 6.6, line 26 defines `pb()` ("propagates before"), which allows a strong fence[4] preceded by `A-cumul` and followed by `B-cumul` to extend the link. Line 27 defines `link()`, which defines a link as being the `A-cumul` relationship concatenated with the `B-cumul` relationship, possibly with any number of `pb()` relationships in between.

Line 30 then defines `mylink` as shorthand for `link(A,B,sf)`. Line 31 defines `gp-link` as an RCU grace period followed by a linking sequence, and line 32 similarly defines `rscs-link` as an RCU read-side critical section followed by a linking sequence. Lines 33–38 defines `rcu-path`, a relation containing sequences of RCU read-side critical sections and RCU grace periods (along with the linking sequences) that have at least as many grace periods as critical sections. Line 39 then prohibits any cycles in the `rcu-path` relation, thus implmenting the RCU counting method [2].

Finally, line 44 performs a consistency check on the `mylink` relation, which should include successive links separated by a strong fence.

Referring back to Listing 6.1, lines 12 and 13 enable easy experimentation. Alternative formulations of the RCU additions to the C11 memory model may be carried out by altering the definitions of `A-cumul` and `B-cumul`.

7 Litmus-Test Filename Decoder Ring

The name of the file is "C-" followed by a litmus-test class name and process descriptors, and ended by ".litmus". Each process descriptor consists of "+" followed by operation designators separated by "-". The operator designators are as follows:

a Acquire load (Linux-kernel `smp_load_acquire()`) or RCU pointer assignment (Linux-kernel `rcu_assign_pointer()`).

addr Address dependency.

ctrl Control dependency.

data Data dependency.

l Lock acquisition (Linux-kernel `spin_lock()`.

L Strongly ordered lock acquisition (Linux-kernel `spin_lock()` followed by `smp_mb__after_spinlock()`).

mb Full memory fence (Linux-kernel `smp_mb()`). Similar to C++ `atomic_thread_fence(memory_order_seq_cst)`.

o "ONCE" access, either Linux-kernel `READ_ONCE()` or `WRITE_ONCE()`, depending on the litmus-test name. These are similar to C++ `volatile` relaxed loads and stores.

r Store release (Linux-kernel `smp_store_release()`) if a store, and the mythical consume load (Linux-kernel `rcu_dereference()`) if a load.

rlk Enter RCU read-side critical section (Linux-kernel `rcu_read_lock()`).

[4] Specifically, `atomic_thread_fence(memory_order_seq_cst)`.

rmb Read memory fence (Linux-kernel `smp_rmb()`).

rulk Exit RCU read-side critical section (Linux-kernel `rcu_read_unlock()`).

sr Wait for relevant RCU readers (Linux-kernel `synchronize_rcu()`).

u Lock release (Linux-kernel `spin_unlock()`.

wmb Write memory fence (Linux-kernel `smp_wmb()`).

References

[1] Alglave, J., Maranget, L., McKenney, P. E., Parri, A., and Stern, A. A formal kernel memory-ordering model (part 1). `https://lwn.net/Articles/718628/`, April 2017.

[2] Alglave, J., Maranget, L., McKenney, P. E., Parri, A., and Stern, A. A formal kernel memory-ordering model (part 2). `https://lwn.net/Articles/720550/`, April 2017.

[3] Boehm, H.-J., Giroux, O., and Vafeiades, V. P0668r1: Revising the C++ memory model. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0668r1.html`, July 2017.

[4] Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K., and Dreyer, D. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, ACM, pp. 618–632.

[5] McKenney, P. E. Read-copy update (RCU) for C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0279r1.pdf`, August 2016.

[6] McKenney, P. E., Michael, M., Wong, M., Muerte, I., OâĂŹDwyer, A., , Hollman, D., Hunter, A., Romer, G., and Roy, L. Proposed RCU C++ API. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0461r2.pdf`, October 2017.

[7] Romer, G., and Hunter, A. An RAII interface for deferred reclamation. Available: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0561r2.html` [Viewed November 13, 2017], October 2017.

[8] Wong, M., Michael, M., and McKenney, P. E. A lock-free concurrency toolkit for deferred reclamation and optimistic speculation. `https://www.youtube.com/watch?v=uhgrD_B1RhQ`, September 2016.