SRCU Grace-Period Ordering

Paul E. McKenney
December 23, 2022

Introduction

This is a brain dump of my understanding of the implementation of SRCU's memory-ordering properties. The implementation's ordering is intended to be redundant, this redundancy being a counterpart of mechanical engineering's safety factors. This means that someone perusing the source code might well come up with different sets of accesses guaranteeing the needed ordering.

This document is *not* part of the official documentation of the Linux-kernel RCU implementation, which is being constructed by Joel Fernandes, Frédéric Weisbecker, and Boqun Feng.

Requirements

Quoting the `synchronize_srcu()` header comment:

```
 * There are memory-ordering constraints implied by synchronize_srcu().
 * On systems with more than one CPU, when synchronize_srcu() returns,
 * each CPU is guaranteed to have executed a full memory barrier since
 * the end of its last corresponding SRCU read-side critical section
 * whose beginning preceded the call to synchronize_srcu().  In addition,
 * each CPU having an SRCU read-side critical section that extends beyond
 * the return from synchronize_srcu() is guaranteed to have executed a
 * full memory barrier after the beginning of synchronize_srcu() and before
 * the beginning of that SRCU read-side critical section.  Note that these
 * guarantees include CPUs that are offline, idle, or executing in user mode,
 * as well as CPUs that are executing in the kernel.
 *
 * Furthermore, if CPU A invoked synchronize_srcu(), which returned
 * to its caller on CPU B, then both CPU A and CPU B are guaranteed
 * to have executed a full memory barrier during the execution of
 * synchronize_srcu().  This guarantee applies even if CPU A and CPU B
 * are the same CPU, but again only if the system has more than one CPU.
 *
 * Of course, these memory-ordering guarantees apply only when
 * synchronize_srcu(), srcu_read_lock(), and srcu_read_unlock() are
 * passed the same srcu_struct structure.
```

To make things easier, let's number and name the different components of this guarantee:

1. The component "`when synchronize_srcu() returns,each CPU is guaranteed to have executed a full memory barrier since the end of its last corresponding SRCU read-side critical section whose beginning preceded the call to synchronize_srcu()`" is called "Since End of Last Critical Section".
2. The component "`each CPU having an SRCU read-side critical section that extends beyond the return from synchronize_srcu() is guaranteed to have executed a full memory barrier after the beginning of synchronize_srcu() and before the beginning of that SRCU read-side critical section`" is called "Before Beginning of Next Critical Section".
3. The component "`if CPU A invoked synchronize_srcu(), which returned to its caller on CPU B, then both CPU A and CPU B are guaranteed to have executed a full memory barrier during the execution of synchronize_srcu()`" is called "During Execution of `synchronize_rcu()`".

Note also that the wording of this comment is sloppy, as in notes to myself rather than a complete mathematical description. In particular, `synchronize_srcu()` is often (but not always!) used as a shorthand for "the SRCU grace period corresponding to a given `synchronize_srcu()`". Also, most of the colloquialisms "before", "after", and friends should be understood to entail memory ordering. To avoid excessive repetition, there will be a "Grace-Period Ordering" component factored out of the above three components.

There will also be a bonus "Forward Progress Considerations" section.

Implementation

Each component is described in the corresponding section below.

Since End of Last Critical Section

On the surface, this one is easy. After all, `srcu_read_unlock()` executes a full memory barrier, so job done, right?

Not quite, given that there are those sneaky colloquialisms "since" and "preceding" hiding in that phrase. This means that the grace-period ordering must have some way of classifying SRCU read-side critical sections that begin before a given grace period and that end after that grace period. Furthermore, the grace-period processing code must provide ordering as well.

In addition, there can be SRCU read-side critical sections that neither begin before a given grace period nor that end after that grace period. These must also be correctly ordered.

Before Beginning of Next Critical Section

The letter of this law is satisfied by the `smp_mb()` executed by `srcu_read_unlock()`. But again, there is that "beyond the return from `synchronize_rcu()`" and "before the beginning of that SRCU read-side critical section. This attain requires that grace-period ordering have some way of classifying SRCU read-side critical sections that begin before a given grace period and that end after that grace period.

During Execution of `synchronize_rcu()`

First, the scheduler provides a full memory barrier on each CPU involved in a given task migration. This is not sufficient in and of itself, but it does guarantee that the effects of each `smp_mb()` in the task executing that `synchronize_srcu()` will still take effect across any task switch.

Let's focus on the end first, because the end is trivially supplied by the `smp_mb()` right at the end of `__synchronize_srcu()`. Note that both `synchronize_srcu()` and `synchronize_srcu_expedited()` invoke `__synchronize_srcu()`.

The full barrier at the beginning of `synchronize_srcu()` can be found by looking at the `__synchronize_srcu()` function's call to `__call_srcu()`, which in turn calls `srcu_gp_start_if_needed()`, which calls `__srcu_read_lock_nmisafe()`, which invokes the required `smp_mb()`.

Grace-Period Ordering

To break up the tedium, this is split into the following sections:

1. Beginning of grace period against grace-period requests.
2. End of grace period against pre-existing critical sections.
3. Grace period beginning and end.
4. Beginning of grace period against surviving critical sections.
5. End of grace period against reclamation.

Beginning of Grace Period Against Grace-Period Requests

The previous section established that `synchronize_srcu()` and friends eventually invoke `__srcu_read_lock_nmisafe()`, which contains the smp_mb() that orders everything following against the pre-grace-period change.

They also invoke `srcu_gp_start_if_needed()`, which in turn invokes `rcu_seq_snap()` on the `srcu_struct` structure's `->srcu_gp_seq` field, all the while holding the `srcu_data` structure's `->lock`, the acquisition of which also implies a full memory barrier.

The `srcu_gp_start_if_needed()` function might need to start a new SRCU grace period, in which case it invokes either `srcu_funnel_gp_start()` or `srcu_funnel_exp_start()`.

But either way, an SRCU grace period will be started, which will invoke `srcu_gp_start()`, which will in turn invoke `rcu_seq_start()` on the `srcu_struct` structure's `->srcu_gp_seq` field. The memory barriers combined with the accesses to this field guarantee that the start of the grace period follows the request for that grace period, be that request `call_srcu()`, `synchronize_srcu()`, or `synchronize_srcu_expedited()`.

Why?

Because the grace-period request did an ordered access to the `srcu_struct` structure's `->srcu_gp_seq` field before the beginning of that grace period did an ordered increment of that same field. Therefore, the entire grace period is ordered after any memory reference preceding any request for that same grace period. QED.

End of Grace Period Against Pre-Existing Critical Sections

Note that while it is just fine to misclassify a limited number of critical sections as being pre-existing, misclassification in the other direction is fatal. To that end, while the beginning of the SRCU grace period is the call to `rcu_seq_start()`, pre-existing SRCU read-side critical sections are recognized all the way down to the `->srcu_lock_count[]` and `->srcu_unlock_count[]` scans.

These scans must be ordered with respect to the formal beginning of the grace period and the read-side primitives. @@@

Grace Period Beginning and End

This one is easy. The grace period began with a call to `rcu_seq_snap()` and ended with a call to `rcu_seq_end()`, both of which do an ordered increment of the `srcu_struct` structure's `->srcu_gp_seq` field. This provides full ordering. However, this ordering must be transmitted on to the post-grace-period code, which depends on the primitive used to request the grace period, which is the topic of the later section "End of grace period against reclamation".

Beginning of Grace Period Against Surviving Critical Sections

Note that while it is just fine to misclassify a limited number of critical sections as being pre-existing, misclassification in the other direction is fatal. @@@

End of Grace Period Against Reclamation

Let's start with `call_srcu()`, which requires that the invocation of the callback be ordered against the end of the grace period.

Ordering stores in readers with loads around synchronize_rcu()

WIP

Forward Progress Considerations

Mind mapping of ordering

https://pasteboard.co/izjxIls6rMCc.jpg

Read-Side Optimizations?

Both `srcu_read_lock()` and `srcu_read_unlock()` invoke `smp_mb()`, which is not the fastest memory-ordering operation in the world. It is hard to imagine being able to safely weaken `srcu_read_lock()` because the counter write must be seen as preceding all accesses in the ensuing read-side critical section. In contrast, downgrading the `srcu_read_unlock()` function's full memory barrier to release semantics is not out of the question, many devils though there might be in the details.

Let's look at the known constraints:

- If the grace-period processing sees the `srcu_read_unlock()` function's unlock increment, then it must also see the lock increment from the corresponding `srcu_read_lock()`. The ordering for this message-passing (MP) pattern is provided by a release operation combined with the `smp_mb()` between the grace-period's unlock and lock summations.
- If the grace-period processing sees the `srcu_read_unlock()` function's unlock increment, then all code following the end of the current grace period must see the effect of all accesses within the corresponding SRCU read-side critical section. This is discussed in the following section.

Critical Section vs. Post-Grace-Period Accesses

Consider the following summary scenario:

```
// Read-side critical section:
```

```
idx = srcu_read_lock(&srcu); // smp_mb()
r1 = READ_ONCE(x);
WRITE_ONCE(y, 1);
WRITE_ONCE(z, 1);
srcu_read_unlock(&srcu, idx); // Release ordering

// Post-grace-period accesses:
WRITE_ONCE(x, 1);
r2 = READ_ONCE(y);
WRITE_ONCE(z, 2);

// Order detection code x:
WRITE_ONCE(x, 2);
smp_mb();
r3 = READ_ONCE(y);

// Order detection code y:
WRITE_ONCE(y, 2);
smp_mb();
r4 = READ_ONCE(z);
```

If a given grace period sees the reader's `srcu_read_lock()`, then it cannot end until it also sees that reader's `srcu_read_unlock()`. The read of the unlock counter is followed by `smp_mb()` A, which orders against subsequent access in the grace-period processing code. This subsequent code includes the calls to `srcu_schedule_cbs_snp()`, which use workqueues to schedule callback invocation. There are accesses to work-queue data structures at both ends of the workqueue handoff, and the `spin_lock_irq_rcu_node(sdp)` within `srcu_invoke_callbacks()` implies a full `smp_mb()` barrier, which forces the callback execution to be fully ordered against the prior grace period.

In the case of `synchronize_srcu()`, he `smp_mb()` at the end of `__synchronize_srcu()` orders the wakeup accesses against those following the return from `synchronize_srcu()`.

Mapping these out for the case where the post-grace-period accesses are in an SRCU callback:

- Read-side critical section: Critical-section accesses -> release -> unlock counter update.
- Grace-period counter access: Unlock counter read -> `smp_mb()` -> workqueue update.
- Workqueue handler invocation: Workqueue read -> `spin_lock_irq_rcu_node(sdp)` use of `smp_mb()` -> callback invocation and hence post-grace-period accesses.

This is a release-acquire chain that connects the critical-section accesses to the post-grace-period accesses, which suffices to provide the required ordering.

Mapping these out for the case where the post-grace-period accesses follow a return from `synchronize_srcu()`:

- Read-side critical section: Critical-section accesses -> release -> unlock counter update.
- Grace-period counter access: Unlock counter read -> `smp_mb()` -> workqueue update.
- Workqueue handler invocation: Workqueue read -> `spin_lock_irq_rcu_node(sdp)` use of `smp_mb()` -> callback invocation's completion-queue runqueue updates.
- Wakeup: Runqueue reads -> –`synchronize_srcu()` use of `smp_mb()` -> post-grace-period accesses.

This is again a release-acquire chain that connects the critical-section accesses to the post-grace-period accesses, which again suffices to provide the required ordering.

However, this is not sufficient because the ordering must be visible to unrelated CPUs, which is the point of the x and y order-detection code. Here is where the LKMM Rule of Thumb #4 comes into play: "At least one full barrier is required between each pair of non-store-to-load links" (see "Is Parallel Programming Hard, And, If So, What Can You Do About It?", Section 15.6). In all cases, this rule of thumb is followed, so the required ordering is provided even to unrelated CPUs.

Therefore, it should be safe to downgrade `srcu_read_unlock()` ordering from `smp_mb()` to some sort of release operation, be it an smp_mb_acq_rel() or a `this_cpu_inc_release()` or whatever.

Documentation

Note well that the synchronize_srcu() header comment states this guarantee:

> `On systems with more than one CPU, when synchronize_srcu() returns, each CPU is guaranteed to have executed a full memory barrier since the end of its last corresponding SRCU read-side critical section whose beginning preceded the call to synchronize_srcu().`

If the `srcu_read_unlock()` function's ordering is downgraded from a full memory barrier to release ordering, then at the very least this comment must change. Preferably there should also be some investigation to see if any code is relying on this guarantee. After all, this guarantee was added to `synchronize_rcu()` because people asked for it. They just might also need it on `synchronize_srcu()`.

What if someone does need this guarantee? One approach is to provide a special `srcu_read_unlock_full()` or some such for them.

Using LKMM litmus tests to build a model of SRCU implementation with GP guarantee verification?

The following 2 litmus tests attempt to build a model of SRCU. We consider scanning done on only 2 CPUs to keep state-space explosion low. To model per-cpu arrays, we use 4 variables (2 CPUs each with 2 IDX values).

Case1: Stores in updater, loads in readers

Litmus test link: https://gist.github.com/joelagnel/3b6537e06a953ff64b205e1019e5afdf
Runs successfully in 20 minutes.

Experiment 1: Delete D/E barriers
Passes, we can clearly see that D/E memory barriers pre and post-flip are not needed for correctness.

Experiment 2: Delete A, keep D/E: Fails.

Experiment 3: Relax the unlock memory barrier (C) in srcu_read_unlock() to smp_store_release()
Passes, Doing this shows that the test still passes.

Experiment 4: Remove “C” completely.
WIP (I will abandon this since I lost the result, and Case 2 fails anyway).

Case2: Stores in readers, loads in updater

Litmus test link: https://gist.github.com/joelagnel/b96706a7fdf70917e54ab7a5e94fe4b7
Runs successfully in 20 seconds.

Experiment 1: Delete D/E barriers
Passes, we can clearly see that D/E memory barriers pre and post-flip are not needed for correctness.

Experiment 2: Remove A, keep D/E : Fails.

Experiment 3: Relax the unlock memory barrier (C) in srcu_read_unlock() to smp_store_release().
Doing this shows that the test still passes.

Experiment 4: Remove “C” completely. : Fails.

Case3: Stores in reader, loads in updater and in post-GP readers

WIP

But is This Worthwhile?

Across Meta’s fleet, `__srcu_read_unlock()` consumes a vanishingly small fraction of the CPU time, as in some orders of magnitude less 0.1%. This is of course absolutely **not** worth going after.

But do other workloads feature more prominent use of this function? Or does it appear on some critical code path in some latency-sensitive workload?