RCU and Unloadable Modules

[Originally published in LWN Jan. 14, 2007: http://lwn.net/Articles/217484/]

RCU (read-copy update) is a synchronization mechanism that can be thought of as a replacement for read-writer locking (among other things), but with very low-overhead readers that are immune to deadlock, priority inversion, and unbounded latency. RCU read-side critical sections are delimited by rcu_read_lock() and rcu_read_unlock(), which, in non-CONFIG_PREEMPT kernels, generate no code whatsoever.

This means that RCU writers are unaware of the presence of concurrent readers, so that RCU updates to shared data must be undertaken quite carefully, leaving an old version of the data structure in place until all pre-existing readers have finished. These old versions are needed because such readers might hold a reference to them. RCU updates can therefore be rather expensive, and RCU is thus best suited for read-mostly situations.

How can an RCU writer possibly determine when all readers are finished, given that readers might well leave absolutely no trace of their presence? There is a synchronize_rcu() primitive that blocks until all pre-existing readers have completed. An updater wishing to delete an element p from a linked list might do the following, while holding an appropriate lock, of course:

```
        list_del_rcu(p);
        synchronize_rcu();
        kfree(p);
```

But the above code cannot be used in IRQ context -- the call_rcu() primitive must be used instead. This primitive takes a pointer to an rcu_head struct placed within the RCU-protected data structure and another pointer to a function that may be invoked later to free that structure. Code to delete an element p from the linked list from IRQ context might then be as follows:

```
        list_del_rcu(p);
        call_rcu(&p->rcu, p_callback);
```

Since call_rcu() never blocks, this code can safely be used from within IRQ context. The function p_callback() might be defined as follows:

```
        static void p_callback(struct rcu_head *rp)
        {
                struct pstruct *p = container_of(rp, struct pstruct, rcu);

                kfree(p);
        }
```

Unloading Modules That Use call_rcu()

But what if p_callback is defined in an unloadable module?

If we unload the module while some RCU callbacks are pending, the CPUs executing these callbacks are going to be severely

disappointed when they are later invoked, as fancifully depicted at
http://lwn.net/images/ns/kernel/rcu-drop.jpg.

We could try placing a synchronize_rcu() in the module-exit code path,
but this is not sufficient. Although synchronize_rcu() does wait for a
grace period to elapse, it does not wait for the callbacks to complete.

One might be tempted to try several back-to-back synchronize_rcu()
calls, but this is still not guaranteed to work. If there is a very
heavy RCU-callback load, then some of the callbacks might be deferred
in order to allow other processing to proceed. Such deferral is required
in realtime kernels in order to avoid excessive scheduling latencies.

rcu_barrier()

We instead need the rcu_barrier() primitive. This primitive is similar
to synchronize_rcu(), but instead of waiting solely for a grace
period to elapse, it also waits for all outstanding RCU callbacks to
complete. Pseudo-code using rcu_barrier() is as follows:

1. Prevent any new RCU callbacks from being posted.
2. Execute rcu_barrier().
3. Allow the module to be unloaded.

Quick Quiz #1: Why is there no srcu_barrier()?

The rcutorture module makes use of rcu_barrier in its exit function
as follows:

```
 1 static void
 2 rcu_torture_cleanup(void)
 3 {
 4   int i;
 5
 6   fullstop = 1;
 7   if (shuffler_task != NULL) {
 8     VERBOSE_PRINTK_STRING("Stopping rcu_torture_shuffle task");
 9     kthread_stop(shuffler_task);
10   }
11   shuffler_task = NULL;
12
13   if (writer_task != NULL) {
14     VERBOSE_PRINTK_STRING("Stopping rcu_torture_writer task");
15     kthread_stop(writer_task);
16   }
17   writer_task = NULL;
18
19   if (reader_tasks != NULL) {
20     for (i = 0; i < nrealreaders; i++) {
21       if (reader_tasks[i] != NULL) {
22         VERBOSE_PRINTK_STRING(
23           "Stopping rcu_torture_reader task");
24         kthread_stop(reader_tasks[i]);
25       }
26       reader_tasks[i] = NULL;
```

```
27      }
28      kfree(reader_tasks);
29      reader_tasks = NULL;
30    }
31    rcu_torture_current = NULL;
32
33    if (fakewriter_tasks != NULL) {
34      for (i = 0; i < nfakewriters; i++) {
35        if (fakewriter_tasks[i] != NULL) {
36          VERBOSE_PRINTK_STRING(
37            "Stopping rcu_torture_fakewriter task");
38          kthread_stop(fakewriter_tasks[i]);
39        }
40        fakewriter_tasks[i] = NULL;
41      }
42      kfree(fakewriter_tasks);
43      fakewriter_tasks = NULL;
44    }
45
46    if (stats_task != NULL) {
47      VERBOSE_PRINTK_STRING("Stopping rcu_torture_stats task");
48      kthread_stop(stats_task);
49    }
50    stats_task = NULL;
51
52    /* Wait for all RCU callbacks to fire. */
53    rcu_barrier();
54
55    rcu_torture_stats_print(); /* -After- the stats thread is stopped! */
56
57    if (cur_ops->cleanup != NULL)
58      cur_ops->cleanup();
59    if (atomic_read(&n_rcu_torture_error))
60      rcu_torture_print_module_parms("End of test: FAILURE");
61    else
62      rcu_torture_print_module_parms("End of test: SUCCESS");
63  }
```

Line 6 sets a global variable that prevents any RCU callbacks from re-posting themselves. This will not be necessary in most cases, since RCU callbacks rarely include calls to call_rcu(). However, the rcutorture module is an exception to this rule, and therefore needs to set this global variable.

Lines 7-50 stop all the kernel tasks associated with the rcutorture module. Therefore, once execution reaches line 53, no more rcutorture RCU callbacks will be posted. The rcu_barrier() call on line 53 waits for any pre-existing callbacks to complete.

Then lines 55-62 print status and do operation-specific cleanup, and then return, permitting the module-unload operation to be completed.

Quick Quiz #2: Is there any other situation where rcu_barrier() might
        be required?

Your module might have additional complications. For example, if your

module invokes call_rcu() from timers, you will need to first cancel all the timers, and only then invoke rcu_barrier() to wait for any remaining RCU callbacks to complete.

Of course, if you module uses call_rcu_bh(), you will need to invoke rcu_barrier_bh() before unloading. Similarly, if your module uses call_rcu_sched(), you will need to invoke rcu_barrier_sched() before unloading. If your module uses call_rcu(), call_rcu_bh(), -and- call_rcu_sched(), then you will need to invoke each of rcu_barrier(), rcu_barrier_bh(), and rcu_barrier_sched().

Implementing rcu_barrier()

Dipankar Sarma's implementation of rcu_barrier() makes use of the fact that RCU callbacks are never reordered once queued on one of the per-CPU queues. His implementation queues an RCU callback on each of the per-CPU callback queues, and then waits until they have all started executing, at which point, all earlier RCU callbacks are guaranteed to have completed.

The original code for rcu_barrier() was as follows:

```
 1 void rcu_barrier(void)
 2 {
 3   BUG_ON(in_interrupt());
 4   /* Take cpucontrol mutex to protect against CPU hotplug */
 5   mutex_lock(&rcu_barrier_mutex);
 6   init_completion(&rcu_barrier_completion);
 7   atomic_set(&rcu_barrier_cpu_count, 0);
 8   on_each_cpu(rcu_barrier_func, NULL, 0, 1);
 9   wait_for_completion(&rcu_barrier_completion);
10   mutex_unlock(&rcu_barrier_mutex);
11 }
```

Line 3 verifies that the caller is in process context, and lines 5 and 10 use rcu_barrier_mutex to ensure that only one rcu_barrier() is using the global completion and counters at a time, which are initialized on lines 6 and 7. Line 8 causes each CPU to invoke rcu_barrier_func(), which is shown below. Note that the final "1" in on_each_cpu()'s argument list ensures that all the calls to rcu_barrier_func() will have completed before on_each_cpu() returns. Line 9 then waits for the completion.

This code was rewritten in 2008 to support rcu_barrier_bh() and rcu_barrier_sched() in addition to the original rcu_barrier().

The rcu_barrier_func() runs on each CPU, where it invokes call_rcu() to post an RCU callback, as follows:

```
 1 static void rcu_barrier_func(void *notused)
 2 {
 3 int cpu = smp_processor_id();
 4 struct rcu_data *rdp = &per_cpu(rcu_data, cpu);
 5 struct rcu_head *head;
 6
 7 head = &rdp->barrier;
 8 atomic_inc(&rcu_barrier_cpu_count);
```

```
 9 call_rcu(head, rcu_barrier_callback);
10 }
```

Lines 3 and 4 locate RCU's internal per-CPU rcu_data structure, which contains the struct rcu_head that needed for the later call to call_rcu(). Line 7 picks up a pointer to this struct rcu_head, and line 8 increments a global counter. This counter will later be decremented by the callback. Line 9 then registers the rcu_barrier_callback() on the current CPU's queue.

The rcu_barrier_callback() function simply atomically decrements the rcu_barrier_cpu_count variable and finalizes the completion when it reaches zero, as follows:

```
 1 static void rcu_barrier_callback(struct rcu_head *notused)
 2 {
 3 if (atomic_dec_and_test(&rcu_barrier_cpu_count))
 4 complete(&rcu_barrier_completion);
 5 }
```

Quick Quiz #3: What happens if CPU 0's rcu_barrier_func() executes immediately (thus incrementing rcu_barrier_cpu_count to the value one), but the other CPU's rcu_barrier_func() invocations are delayed for a full grace period? Couldn't this result in rcu_barrier() returning prematurely?

rcu_barrier() Summary

The rcu_barrier() primitive has seen relatively little use, since most code using RCU is in the core kernel rather than in modules. However, if you are using RCU from an unloadable module, you need to use rcu_barrier() so that your module may be safely unloaded.

Answers to Quick Quizzes

Quick Quiz #1: Why is there no srcu_barrier()?

Answer: Since there is no call_srcu(), there can be no outstanding SRCU callbacks. Therefore, there is no need to wait for them.

Quick Quiz #2: Is there any other situation where rcu_barrier() might be required?

Answer: Interestingly enough, rcu_barrier() was not originally implemented for module unloading. Nikita Danilov was using RCU in a filesystem, which resulted in a similar situation at filesystem-unmount time. Dipankar Sarma coded up rcu_barrier() in response, so that Nikita could invoke it during the filesystem-unmount process.

Much later, yours truly hit the RCU module-unload problem when implementing rcutorture, and found that rcu_barrier() solves this problem as well.

Quick Quiz #3: What happens if CPU 0's rcu_barrier_func() executes immediately (thus incrementing rcu_barrier_cpu_count to the value one), but the other CPU's rcu_barrier_func() invocations are delayed for a full grace period? Couldn't this result in rcu_barrier() returning prematurely?

Answer: This cannot happen. The reason is that on_each_cpu() has its last argument, the wait flag, set to "1". This flag is passed through to smp_call_function() and further to smp_call_function_on_cpu(), causing this latter to spin until the cross-CPU invocation of rcu_barrier_func() has completed. This by itself would prevent a grace period from completing on non-CONFIG_PREEMPT kernels, since each CPU must undergo a context switch (or other quiescent state) before the grace period can complete. However, this is of no use in CONFIG_PREEMPT kernels.

Therefore, on_each_cpu() disables preemption across its call to smp_call_function() and also across the local call to rcu_barrier_func(). This prevents the local CPU from context switching, again preventing grace periods from completing. This means that all CPUs have executed rcu_barrier_func() before the first rcu_barrier_callback() can possibly execute, in turn preventing rcu_barrier_cpu_count from prematurely reaching zero.

Currently, -rt implementations of RCU keep but a single global queue for RCU callbacks, and thus do not suffer from this problem. However, when the -rt RCU eventually does have per-CPU callback queues, things will have to change. One simple change is to add an rcu_read_lock() before line 8 of rcu_barrier() and an rcu_read_unlock() after line 8 of this same function. If you can think of a better change, please let me know!