Priority-Boosting RCU Read-Side Critical Sections

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

April 16, 2007

1 Introduction

Read-copy update (RCU) is a synchronization API that is sometimes used in place of reader-writer locks. RCU's read-side primitives offer extremely low overhead and deterministic execution time. These properties imply that RCU updaters cannot block RCU readers, which means that RCU updates can be expensive, as they must leave old versions of the data structure in place to accommodate pre-existing readers. Furthermore, these old versions must be reclaimed after all pre-existing readers complete. The Linux kernel offers a number of RCU implementations, the first such implementation being called "Classic RCU".

The RCU implementation for the -rt patchset is unusual in that it permits read-side critical sections to be blocked waiting for locks and due to preemption. If these critical sections are blocked for too long, grace periods will be stalled, and the amount of memory awaiting the end of a grace period will continually increase, eventually resulting in an out-of-memory condition. This theoretical possibility was apparent from the start, but when Trevor Woerner actually made it happen, it was clear that something needed to be done. Because priority boosting is used in locking, it seemed natural to apply it to realtime RCU.

Unfortunately, the priority-boosting algorithm used for locking could not be applied straightforwardly to RCU because this algorithm uses locking, and the whole point of RCU is to *avoid* common-case use of such heavy-weight operations in read-side primitives. In fact, RCU's read-side primitives need to avoid common-case use of *all* heavyweight operations, including atomic instructions, memory barriers, and cache misses. Therefore, bringing priority boosting to RCU turned out to be rather challenging, not because the eventual solution is all that complicated, but rather due to the large number of seductive but subtly wrong almost-solutions.

This document describes a way of providing lightweight priority boosting to RCU, and also describes several of the number of seductive but subtlely wrong almost-solutions.

2 Approaches

This paper describes three approaches to priority-boosting blocked RCU read-side critical sections. The first approach minimizes scheduler-path overhead and uses locking on non-fastpaths to decrease complexity. The second approach is similar to the first, and was in fact a higher-complexity intermediate point on the path to the first approach. The third approach uses a per-task lock solely for its priority-inheritance properties, which introduces the overhead of acquiring this lock into the scheduler path, but avoids adding an "RCU boost" component to the priority calculations. Unfortunately, this third approach also cannot be made to reliably boost tasks blocked in RCU read-side critical sections, so the first approach should be used to the exclusion of the other

two. Each of these approaches is described in a following section, after which is a section enumerating other roads not taken.

3 RCU Explicit Priority Boosting

The solution described in this paper makes use of a separate RCU-booster task that monitors per-CPU arrays of lists of target tasks that have been blocked while in an RCU read-side critical section. Overhead is incurred only when such blocking occurs, permitting the RCU read-side-acquisition primitive (e.g., `rcu_read_lock()`) to contain exactly the same sequence of instructions contained in its non-boosted counterpart.

3.1 Data Structures

Each element of each per-CPU array is a structure named `struct rcu_boost_dat` as shown in Figure 1. The `rbs_mutex` field guards all fields in the structure, `rbs_toboost` is a list containing target tasks that are candidates for boosting, and `rbs_boosted` is a list containing target tasks that have already been boosted. The rest of the fields are statistics: `rbs_blocked` counts the number of RCU read-side critical sections that were blocked (whether once or multiple times), `rbs_boost_attempt` counts the number of tasks that the RCU-booster task has attempted to boost, `rbs_boost` counts the number of such attempts in which boosting was accomplished, `rbs_unlock` counts the number of outermost `rcu_read_unlock()` operations that end a blocked RCU read-side critical section, and `rbs_unboosted` counts the number of tasks whose that needed to be unboosted by `rcu_read_unlock()`. The `rbs_stats[]` array tracks the number of transitions due to each event from each state.

Figure 2 shows the `task_struct` fields that are used in RCU priority boosting. The `rcub_rbdp` is a pointer to the `rcu_boost_dat` struct on which this task is enqueued, `rcub_state` holds the RCU priority-boost state for this task, and `rcub_entry` is

```
 1 struct rcu_boost_dat {
 2   raw_spinlock_t rbs_mutex;
 3   struct list_head rbs_toboost;
 4   struct list_head rbs_boosted;
 5   long rbs_blocked;
 6   long rbs_boost_attempt;
 7   long rbs_boost;
 8   long rbs_unlock;
 9   long rbs_unboosted;
10   long rbs_stats[][];
11 }
```

Figure 1: RCU-Boost Data Structure

the list entry used to enqueue this task on either the `rbs_toboost` or the `rbs_boosted` lists.

```
1   struct rcu_boost_dat *rcub_rbdp;
2   enum rcu_boost_state rcub_state;
3   struct list_head rcub_entry;
```

Figure 2: RCU-Boost Task-Struct Fields

A schematic of the organization of the `rcu_boost_dat` data structure is shown in Figure 3. The `rbs_toboost` fields are represented by the upper set of elements, and the `rbs_boosted` fields are represented by the lower set of elements. These elements are indexed in a circular fashion based on the value of the global index, which is named `rcu_boost_idx`. Please note that corresponding elements in the upper and lower sets in the figure are guarded by the same spinlock, the `rbs_mutex` field. Use of this index eliminates the need to physically move target tasks from one locking domain to another, thus guaranteeing that a given task is subject to the same lock throughout, eliminating the need for latency-prone retry-style lock acquisition.

For a given CPU, the indexed element indicates in which list to place target tasks that have just blocked within an RCU read-side critical section. When a given target task exits its outermost RCU read-side critical section that was blocked, that task removes itself from whatever list it was added to, and also unboosts itself if need be.

A separate RCU priority-boosting task increments the index periodically (modulo the size of the array), resulting in the configuration shown in Figure 4. After each such increments, this RCU-boost task boosts

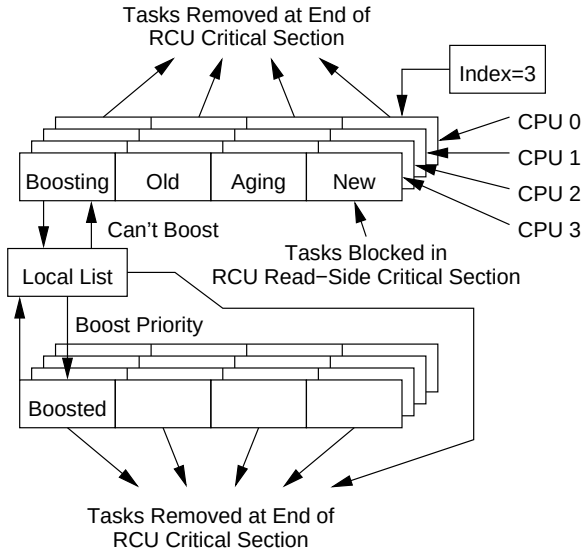

Figure 3: RCU Priority-Boosting Data Structures

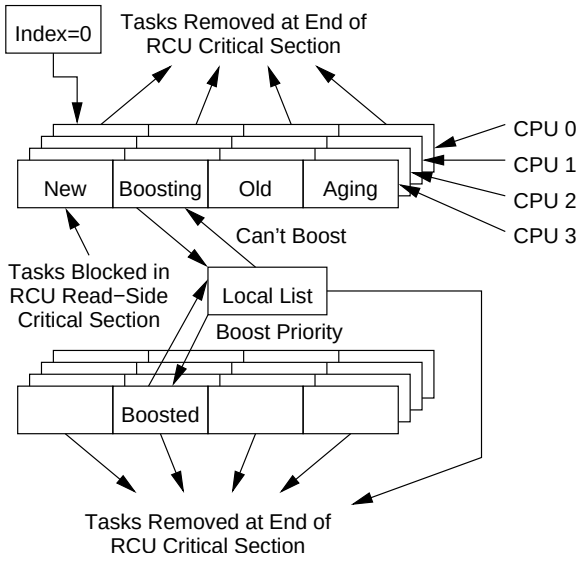

Figure 4: RCU Priority-Boosting Data Structures After Increment

the priority of those (hopefully few) target tasks that have remained for a full time period, as well as any previously boosted tasks that still remain in the list. This reboosting is performed to allow for the possibility that the system administrator changed the priority of the RCU-booster task since the last time those tasks were boosted. Such boosting might well be attempted concurrently with the target task removing itself from the list. Much care is required in order to avoid boosting a target task just after it removes itself from its list. Failure to avoid this scenario could result in an otherwise low-priority task remaining boosted indefinitely, in turn causing other high-priority realtime tasks to miss their deadlines.

The state machine described in the next section prevents such failure scenarios from occurring.

3.2 State Diagram

Each task has an associated RCU-booster state, which can take on the values shown in Figure 5. Tasks in the `RCU_BOOST_BLOCKED` state are linked into the uppermost of the two sets of lists shown in Figure 3, while tasks in the `RCU_BOOSTED` state are linked into the lower of these two sets of lists, and tasks in the `RCU_BOOST_IDLE` state are not linked into either set of lists. The black state transitions are traversed by the task, while the red transitions are traversed by the RCU-booster task.

All priority boosting is carried out by the RCU-booster task, while all priority unboosting is carried out by the target task. This approach ensures that unboosting is exact, preventing low-priority tasks from running at high priority any longer than necessary.

3.3 Per-State Details

The purpose of the individual states are as follows:

- `RCU_BOOST_IDLE`: This is the initial state. A target task resides in this state when not in an RCU read-side critical section, or when in an RCU read-side critical section that has not yet blocked.
- `RCU_BOOST_BLOCKED`: The target task has blocked while in an RCU read-side critical sec-

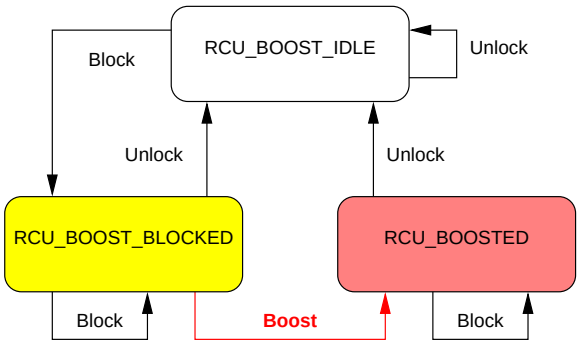

Figure 5: RCU Priority-Boosting State Diagram

```
 1 struct rcu_boost_dat *rcu_rbd_new(void)
 2 {
 3   int cpu = raw_smp_processor_id();
 4   int idx = rcu_boost_idx;
 5
 6   smp_read_barrier_depends(); barrier();
 7   if (unlikely(idx < 0))
 8     return (NULL);
 9   return &per_cpu(rcu_boost_dat, cpu)[idx];
10 }
11
12 struct rcu_boost_dat *rcu_rbd_boosting(int cpu)
13 {
14   int idx = (rcu_boost_idx + 1) & (RCU_BOOST_ELEMENTS - 1);
15
16   return &per_cpu(rcu_boost_dat, cpu)[idx];
17 }
```

Figure 6: Data-Structure Access Helper Functions

tion.

- `RCU_BOOSTED`: The RCU-booster task has completed boosting the target task's priority.

3.4 Events

The important events for a target task are (1) being blocked in an RCU read-side critical section and (2) completing a previously blocked RCU read-side critical section. When a target task is blocked in an RCU read-side critical section, it always adds itself to the current list of its per-CPU array. Conversely, when a target task completes a previously blocked RCU read-side critical section, it removes itself from this list. If its priority has been boosted, it also unboosts itself.

The important event for the RCU-booster task is boosting a target task's priority.

3.5 Implementation

3.5.1 Helper Functions

Figure 6 shows functions used to access the proper element of the arrays used to track tasks that are candidates for RCU priority boosting (see Figure 3 for a schematic of the overall data structure and Figure 1 for the C-language definition of each element of the per-CPU arrays). The `rcu_rbd_new()` function is used by newly blocked tasks adding themselves to the data structure, while the `rcu_rbd_boosting()` function is used by the RCU-booster task to locate tasks in need of boosting.

The `rcu_rbd_new()` function returns the `rcu_boost_dat` element be used for newly blocked tasks adding themselves to the data structure. Lines 3 and 4 pick up the CPU ID and the current value of the index, respectively. Line 6 issues any memory barriers needed on Alpha and prevents the compiler from optimizing bugs into this function on other platforms. Line 7 checks to see if this function is being called before the RCU-booster task has completed initialization, and, if so, line 8 returns. Otherwise, line 9 uses the CPU and index to select the right `rcu_boost_dat` structure to queue on.

The `rcu_rbd_boosting` is simpler because it is invoked only from the RCU-booster task, and therefore cannot run concurrently with a counter increment (although this could change if there are ever multiple RCU-booster tasks). Line 14 selects the index that was least-recently the target of tasks newly blocked in RCU read-side critical sections, and line 16 uses the index and the specified CPU to select the right `rcu_boost_dat` structure to boost from.

The `rcu_boost_prio()` and `rcu_unboost_prio()` functions shown in Figure 7 boost and unboost the specified task's RCU priority.

Lines 6-10 of `rcu_boost_prio()` get the priority of the current (RCU-booster) task, setting the target task's `rcu_prio` field to one notch less-favored prior-

```
 1 static void rcu_boost_prio(struct task_struct *taskp)
 2 {
 3   unsigned long oldirq;
 4   int rcuprio;
 5
 6   spin_lock_irqsave(&current->pi_lock, oldirq);
 7   rcuprio = rt_mutex_getprio(current) + 1;
 8   if (rcuprio >= MAX_USER_RT_PRIO)
 9     rcuprio = MAX_USER_RT_PRIO - 1;
10   spin_unlock_irqrestore(&current->pi_lock, oldirq);
11   spin_lock_irqsave(&taskp->pi_lock, oldirq);
12   if (taskp->rcu_prio != rcuprio) {
13     taskp->rcu_prio = rcuprio;
14     if (taskp->rcu_prio != taskp->prio)
15       rt_mutex_setprio(taskp, taskp->rcu_prio);
16   }
17   spin_unlock_irqrestore(&taskp->pi_lock, oldirq);
18 }
19
20 static void rcu_unboost_prio(struct task_struct *taskp)
21 {
22   int nprio;
23   unsigned long oldirq;
24
25   spin_lock_irqsave(&taskp->pi_lock, oldirq);
26   taskp->rcu_prio = MAX_PRIO;
27   nprio = rt_mutex_getprio(taskp);
28   if (nprio > taskp->prio)
29     rt_mutex_setprio(taskp, nprio);
30   spin_unlock_irqrestore(&taskp->pi_lock, oldirq);
31 }
```

Figure 7: Priority-Boost Helper Functions

ity if possible, and to the least-favored realtime priority otherwise. The `rt_mutex_getprio()` primitive is used to actually obtain this priority. Lines 11-17 boost the target task's priority, with line 12 checking to see if the task has already been RCU-boosted to the desired priority. If not, line 13 updates the task's `rcu_prio` field, line 14 checks to see if the task is already running at the desired priority (perhaps due to lock-based boosting), and, if not, line 15 does the actual priority change. Note that this function is capable of decreasing a task's priority, as will be described below.

Lines 25-30 of `rcu_unboost_prio()` unboost the target task, again using `rt_mutex_getprio()` and `rt_mutex_setprio()` to manipulate the priorities. Line 26 updates the task's `rcu_prio` field to prevent any future priority calculations from adding an RCU component to the priority. Line 28 checks to see if the task is already running at a less-favorable priority before actually deboosting on line 29.

Both functions hold the given task's `pi_lock` in order to properly synchronize with other changes to that task's priority. In addition, the `rt_mutex_getprio()` and `rt_mutex_setprio()` primitives have been modified to take the task's `rcu_prio` field into account in the priority calculations, ensuring that possible lock-based priority deboosts will not remove the RCU priority boost.

3.5.2 Blocking Within an RCU Read-Side Critical Section

The scheduler contains a call to `rcu_preempt_boost`, which is shown on lines 1-5 of Figure 8. This macro checks to see if the current task is in an RCU read-side critical section, and, if so, invokes the `__rcu_preempt_boost()` function to place the calling task on the priority-boost lists.

The `__rcu_preempt_boost()` function runs with irqs disabled, and is potentially invoked with irqs already disabled. Line 12 disables irqs, and line 13 identifies the `struct rcu_boost_dat` that is to be used Line 14 then checks to see whether the RCU-booster task has started, and lines 15-18 restore irqs and returns if so.

Otherwise, line 19 acquires the lock. Note that

```
 1 #define rcu_preempt_boost() \
 2   do { \
 3     if (unlikely(current->rcu_read_lock_nesting > 0)) \
 4       __rcu_preempt_boost(); \
 5   } while (0)
 6
 7 void __rcu_preempt_boost(void)
 8 {
 9   struct rcu_boost_dat *rbdp;
10   unsigned long oldirq;
11
12   local_irq_save(oldirq);
13   rbdp = rcu_rbd_new();
14   if (rbdp == NULL) {
15     local_irq_restore(oldirq);
16     printk("Preempted RCU too early.\n");
17     return;
18   }
19   spin_lock(&rbdp->rbs_mutex);
20   rbdp->rbs_blocked++;
21   rcu_boost_dat_stat_block(rbdp, current->rcub_state);
22   if (current->rcub_state != RCU_BOOST_IDLE) {
23     spin_unlock_irqrestore(&rbdp->rbs_mutex, oldirq);
24     return;
25   }
26   current->rcub_state = RCU_BOOST_BLOCKED;
27   list_add_tail(&current->rcub_entry, &rbdp->rbs_toboost);
28   current->rcub_rbdp = rbdp;
29   spin_unlock_irqrestore(&rbdp->rbs_mutex, oldirq);
30 }
```

Figure 8: rcu_preempt_boost()

the index can change during this time, but because the index must be incremented three times in order for the RCU-booster task to get to this entry, this situation is unlikely to result in premature boosting. Lines 20-21 gather statistics.

Lines 22-25 check to see if this task is already on the priority-boost lists, and if so, restores irqs and returns. Otherwise, line 26 updates the task's state to indicate that this task has blocked in its current RCU read-side critical section, line 27 adds it to the appropriate priority-boost list, line 28 caches a pointer to the list in the task structure, and line 29 releases the lock and restores irqs.

3.5.3 Boosting The Priority of a List of Tasks

The `rcu_boost_one_reader_list` function shown in Figure 9 is invoked by the RCU-booster task to priority-boost all tasks still remaining on the specified `rcu_boost_dat` structure. The boosting is done under the protection of this structure's mutex, but this mutex is periodically dropped to allow the RCU-

```
 1 static void rcu_boost_one_reader_list(struct rcu_boost_dat *rbdp)
 2 {
 3   LIST_HEAD(list);
 4   unsigned long oldirq;
 5   struct task_struct *taskp;
 6
 7   spin_lock_irqsave(&rbdp->rbs_mutex, oldirq);
 8   list_splice_init(&rbdp->rbs_toboost, &list);
 9   list_splice_init(&rbdp->rbs_boosted, &list);
10   while (!list_empty(&list)) {
11     spin_unlock_irqrestore(&rbdp->rbs_mutex, oldirq);
12     schedule_timeout_uninterruptible(1);
13     spin_lock_irqsave(&rbdp->rbs_mutex, oldirq);
14     if (list_empty(&list))
15       break;
16     taskp = list_entry(list.next, typeof(*taskp), rcub_entry);
17     list_del_init(&taskp->rcub_entry);
18     rbdp->rbs_boost_attempt++;
19     if (taskp->rcub_state == RCU_BOOST_IDLE) {
20       list_add_tail(&taskp->rcub_entry, &rbdp->rbs_toboost);
21       rcu_boost_dat_stat_boost(rbdp, taskp->rcub_state);
22       continue;
23     }
24     rcu_boost_prio(taskp);
25     taskp->rcub_state = RCU_BOOSTED;
26     rbdp->rbs_boost++;
27     rcu_boost_dat_stat_boost(rbdp, RCU_BOOST_BLOCKED);
28     list_add_tail(&taskp->rcub_entry, &rbdp->rbs_boosted);
29   }
30   spin_unlock_irqrestore(&rbdp->rbs_mutex, oldirq);
31 }
```

Figure 9: rcu_boost_one_reader_list()

booster task to sleep, thus avoiding imposing excessive scheduling latencies on realtime tasks. Lines 8 and 9 pull the contents of the `rcu_boost_dat` structure's two lists onto a local list. This has the effect of reboosting tasks, which is useful in case the system administrator manually increased the RCU-booster task's priority since the previous boost. Line 10 sequences through the list, and lines 11-13 sleep as noted earlier. Lines 14-15 recheck the list to allow for the possibility of the tasks having exited their RCU read-side critical sections in the meantime, thus removing themselves from the list.

Line 16-17 removes the first task on the list, and line 18 updates statistics. Lines 19-23 reject tasks that are in the wrong state (but counts them), and puts them back on the `rbs_toboost` list. Line 24 boosts the task, line 25 updates state to indicate that this task has had its priority boosted, lines 26-27 accumulate statistics, and line 28 puts the task on the `rbs_boosted` list. If the task is still present after the index has been incremented four more times, it may be boosted again, as noted above, allowing any recent changes in the priority of the RCU-booster task to be taken into account.

3.5.4 Sequencing Through Lists of Tasks

```
 1 static int rcu_booster(void *arg)
 2 {
 3   int cpu;
 4   struct sched_param sp;
 5
 6   sp.sched_priority = PREEMPT_RCU_BOOSTER_PRIO;
 7   sched_setscheduler(current, SCHED_RR, &sp);
 8   current->flags |= PF_NOFREEZE;
 9   do {
10     rcu_boost_idx = (rcu_boost_idx + 1) % RCU_BOOST_ELEMENTS;
11     for_each_possible_cpu(cpu) {
12       rcu_boost_one_reader_list(rcu_rbd_boosting(cpu));
13     }
14     schedule_timeout_uninterruptible(HZ / 100);
15     rcu_boost_dat_stat_print();
16   } while (!kthread_should_stop());
17   return 0;
18 }
```

Figure 10: rcu_booster()

The `rcu_booster()` function shown in Figure 10 periodically cycles through the lists of tasks that may be in need of priority boosting, running as a kernel thread. Lines 6-8 set this task to its default realtime priority, and prevent it from interfering with suspend processing. The loop starting at line 9 makes one pass through each CPU's candidate RCU-boost target tasks, with line 10 advancing the index so as to "age" all tasks that have blocked while in their current RCU read-side critical section. This advancing of the index will require special coordination should it ever be necessary to have multiple RCU-booster tasks. Lines 11-13 invoke `rcu_boost_one_reader_list()` on each CPU's most-aged `rcu_boost_dat` structure in order to boost any tasks that have been blocked for a long time in an RCU read-side critical section for each CPU. Line 14 waits for about 10 milliseconds, which means that tasks must remain in their RCU read-side critical sections for at least 30 milliseconds to become candidates for boosting – *and* they have to have blocked at least once during that time. This seems like a good default setting, but more experience is required to determine what really is appropriate. Line 15 prints statistics if the `PREEMPT_RCU_BOOST_STATS` config parameter is set.

3.5.5 Unboosting

Figure 11 shows `rcu_read_unlock_unboost()`, which is invoked from `rcu_read_unlock()` to unboost the current task's priority if needed. The call from `rcu_read_unlock()` is placed so that `rcu_read_unlock_unboost()` is only invoked from the end of the outermost of a set of nested RCU read-side critical sections. This function is an inlineable wrapper around `__rcu_read_unlock_unboost()`, which it invokes only if the current task was blocked during the preceding RCU read-side critical section.

This same figure also shows the `__rcu_read_unlock_unboost()` helper function starting at line 7. Line 12 retrieves the pointer to the `rcu_boost_dat` that was cached by line 28 of `__rcu_preempt_boost()` in Figure 8. Line 13 (back in Figure 11) then acquires the corresponding lock. Line 15 removes this task from whatever list it is on, line 16 counts the unlock, and line 17 NULLs out the cached pointer. Line 19 accumulates statistics, and lines 20-23 unboost the

```
 1 static inline void rcu_read_unlock_unboost(void)
 2 {
 3   if (unlikely(current->rcub_state != RCU_BOOST_IDLE))
 4     __rcu_read_unlock_unboost();
 5 }
 6
 7 static void __rcu_read_unlock_unboost(void)
 8 {
 9   unsigned long oldirq;
10   struct rcu_boost_dat *rbdp;
11
12   rbdp = current->rcub_rbdp;
13   spin_lock_irqsave(&rbdp->rbs_mutex, oldirq);
14
15   list_del_init(&current->rcub_entry);
16   rbdp->rbs_unlock++;
17   current->rcub_rbdp = NULL;
18
19   rcu_boost_dat_stat_unlock(rbdp, current->rcub_state);
20   if (current->rcub_state == RCU_BOOSTED) {
21     rcu_unboost_prio(current);
22     rbdp->rbs_unboosted++;
23   }
24   current->rcub_state = RCU_BOOST_IDLE;
25   spin_unlock_irqrestore(&rbdp->rbs_mutex, oldirq);
26 }
```

Figure 11: rcu_read_unlock_unboost()

task's priority and count the unboost, but only if the task was boosted. Line 24 sets the task's state to indicate that it is no longer in an RCU read-side critical section in which it has blocked, and, finally, line 25 releases the lock.

3.5.6 Initialization

Figure 12 shows the early-boot initialization code. This is quite straightforward, aside from the memory barrier on line 29 to prevent other CPUs from seeing `rcu_boost_idx` with a non-negative value but uninitialized values for the structures. Just in case early boot processing ever goes SMP!

However, `init_rcu_boost_early()` is called early in the boot process, as the name suggests – early enough, in fact, that the scheduler is not yet functional. Therefore, the creation of the RCU-booster task must be deferred until later, when the scheduler is functional. Figure 13 shows the `init_rcu_boost_late()` function, which simply spawns the RCU-booster task and then returns.

```
 1 static void init_rcu_boost_early(void)
 2 {
 3   struct rcu_boost_dat *rbdp;
 4   int cpu;
 5   int i;
 6
 7   for_each_possible_cpu(cpu) {
 8     rbdp = per_cpu(rcu_boost_dat, cpu);
 9     for (i = 0; i < RCU_BOOST_ELEMENTS; i++) {
10       rbdp[i].rbs_mutex =
11         RAW_SPIN_LOCK_UNLOCKED(rbdp[i].rbs_mutex);
12       INIT_LIST_HEAD(&rbdp[i].rbs_toboost);
13       INIT_LIST_HEAD(&rbdp[i].rbs_boosted);
14       rbdp[i].rbs_blocked = 0;
15       rbdp[i].rbs_boost_attempt = 0;
16       rbdp[i].rbs_boost = 0;
17       rbdp[i].rbs_unlock = 0;
18       rbdp[i].rbs_unboosted = 0;
19 #ifdef CONFIG_PREEMPT_RCU_BOOST_STATS
20       {
21         int j, k;
22
23         for (j = 0; j < N_RCU_BOOST_DAT_EVENTS; j++)
24           for (k = 0; k <= N_RCU_BOOST_STATE; k++)
25             rbdp[i].rbs_stats[j][k] = 0;
26       }
27 #endif /* #ifdef CONFIG_PREEMPT_RCU_BOOST_STATS */
28     }
29     smp_wmb();
30     rcu_boost_idx = 0;
31   }
32 }
```

Figure 12: init_rcu_boost_early()

```
 1 void init_rcu_boost_late(void)
 2 {
 3   int i;
 4
 5   printk(KERN_ALERT "Starting RCU priority booster\n");
 6   rcu_boost_task = kthread_run(rcu_booster, NULL, "RCU Prio Booster");
 7   if (IS_ERR(rcu_boost_task)) {
 8     i = PTR_ERR(rcu_boost_task);
 9     printk(KERN_ALERT
10            "Unable to create RCU Priority Booster, errno %d\n", -i);
11     rcu_boost_task = NULL;
12   }
13 }
```

Figure 13: init_rcu_boost_late()

3.5.7 Statistics Gathering and Output

```
 1 static inline void
 2 rcu_boost_dat_stat(struct rcu_boost_dat *rbdp,
 3                    int event,
 4                    enum rcu_boost_state oldstate)
 5 {
 6   if (oldstate >= RCU_BOOST_IDLE &&
 7       oldstate <= RCU_BOOSTED) {
 8     rbdp->rbs_stats[event][oldstate]++;
 9   } else {
10     rbdp->rbs_stats[event][N_RCU_BOOST_STATE]++;
11   }
12 }
13
14 #define rcu_boost_dat_stat_block(rbdp, oldstate) \
15   rcu_boost_dat_stat(rbdp, RCU_BOOST_DAT_BLOCK, oldstate)
16 #define rcu_boost_dat_stat_boost(rbdp, oldstate) \
17   rcu_boost_dat_stat(rbdp, RCU_BOOST_DAT_BOOST, oldstate)
18 #define rcu_boost_dat_stat_unlock(rbdp, oldstate) \
19   rcu_boost_dat_stat(rbdp, RCU_BOOST_DAT_UNLOCK, oldstate)
```

Figure 14: rcu_boost_dat_stat()

Figure 14 shows the statistics-accumulation functions and macros, which are defined only if the `PREEMPT_RCU_BOOST_STATS` configuration parameter is set. The `rcu_boost_dat_stat()` function increments the element of the `rbs_stats[]` array selected by the event and state, but only for valid state values. Invalid state values cause one of the special invalid-state elements to be incremented, depending on the event. The `rcu_boost_dat_stat_block()`, `rcu_boost_dat_stat_boost()`, and `rcu_boost_dat_stat_unlock()` macros serve as short-hand wrappers for the `rcu_boost_dat_stat` function.

Figure 15 shows arrays used to assist in `printk()`-ing of statistics. The labels defined in `rcu_boost_state_event[]` serve as output line labels, while the characters defined in `rcu_boost_state_error[]` are used to note invalid state transitions. The "?" character indicates a completely invalid state, either because the state itself is undefined or because there is an enclosing check eliminating it, while the "!" character indicates a list-manipulation error. The space character indicates a valid state transition.

```
 1 static char *rcu_boost_state_event[] = {
 2   "block:  ",
 3   "boost:  ",
 4   "unlock: ",
 5 };
 6
 7 static char *rcu_boost_state_error[] = {
 8         /*ibBe*/
 9   "   ?",  /* block */
10   "!  ?",  /* boost */
11   "?  ?",  /* unlock */
12 };
```

Figure 15: Tables for rcu_boost_dat_stat_print()

Figure 16 shows accumulation and printing of statistics. This function refuses to take action unless sufficient time has elapsed since the last time it printed statistics, as can be seen from lines 12-14 and line 56. Locking will be required should multiple RCU-booster tasks ever become necessary. Lines 15-39 sum the corresponding statistical counters from all of the `rcu_boost_dat` structures, and lines 40-55 print out the sums.

```
 1 static void rcu_boost_dat_stat_print(void)
 2 {
 3   char buf[N_RCU_BOOST_STATE * (sizeof(long) * 3 + 2) + 2];
 4   int cpu;
 5   int event;
 6   int i;
 7   static time_t lastprint = 0;
 8   struct rcu_boost_dat *rbdp;
 9   int state;
10   struct rcu_boost_dat sum;
11
12   if (xtime.tv_sec - lastprint <
13       CONFIG_PREEMPT_RCU_BOOST_STATS_INTERVAL)
14     return;
15   sum.rbs_blocked = 0;
16   sum.rbs_boost_attempt = 0;
17   sum.rbs_boost = 0;
18   sum.rbs_unlock = 0;
19   sum.rbs_unboosted = 0;
20   for_each_possible_cpu(cpu)
21     for (i = 0; i < RCU_BOOST_ELEMENTS; i++) {
22       rbdp = per_cpu(rcu_boost_dat, cpu);
23       sum.rbs_blocked += rbdp[i].rbs_blocked;
24       sum.rbs_boost_attempt += rbdp[i].rbs_boost_attempt;
25       sum.rbs_boost += rbdp[i].rbs_boost;
26       sum.rbs_unlock += rbdp[i].rbs_unlock;
27       sum.rbs_unboosted += rbdp[i].rbs_unboosted;
28     }
29   for (event = 0; event < N_RCU_BOOST_DAT_EVENTS; event++)
30     for (state = 0; state <= N_RCU_BOOST_STATE; state++) {
31       sum.rbs_stats[event][state] = 0;
32       for_each_possible_cpu(cpu) {
33         for (i = 0; i < RCU_BOOST_ELEMENTS; i++) {
34           sum.rbs_stats[event][state]
35               += per_cpu(rcu_boost_dat,
36                    cpu)[i].rbs_stats[event][state];
37         }
38       }
39     }
40   printk(KERN_ALERT
41          "rcu_boost_dat: idx=%d "
42          "b=%ld ul=%ld ub=%ld boost: a=%ld b=%ld\n",
43          rcu_boost_idx,
44          sum.rbs_blocked, sum.rbs_unlock, sum.rbs_unboosted,
45          sum.rbs_boost_attempt, sum.rbs_boost);
46   for (event = 0; event < N_RCU_BOOST_DAT_EVENTS; event++) {
47     i = 0;
48     for (state = 0; state <= N_RCU_BOOST_STATE; state++) {
49       i += sprintf(&buf[i], " %ld%c",
50              sum.rbs_stats[event][state],
51              rcu_boost_state_error[event][state]);
52     }
53     printk(KERN_ALERT "rcu_boost_dat %s %s\n",
54            rcu_boost_state_event[event], buf);
55   }
56   lastprint = xtime.tv_sec;
57 }
```

Figure 16: rcu_boost_dat_stat_print()

3.5.8 Possible Future Enhancements

The act of getting any piece of software working (or even sort of working) almost always immediately suggested enhancements, and RCU priority boosting is no exception. This following list includes a few possibilities.

1. Boosting upon OOM. This is currently to be accomplished by having the RCU-booster task more aggressively boost when it becomes aware of OOM conditions, for example, reducing or omitting timed sleeps. Alternatively, one could activate the "canary" mechanism upon OOM, which could downgrade realtime tasks to non-realtime status, allowing the normal aging mechanisms to boost any blocked task that might be holding up a grace period.

2. Successive boosting to ever-higher priorities. This will likely be required in cases where the system administrator manually increases the priority of the RCU-booster task. The intent in this case is no doubt to kick some laggart RCU read-side critical section, which won't happen if the corresponding task has already been boosted. This is deferred until needed.

3. Use of multiple per-CPU or per-NUMA-node RCU booster tasks. This will undoubtedly be required for large systems, but is deferred until needed.

4 RCU Priority Boosting With Minimal Scheduler-Path Overhead

The solution described in this paper makes use of a separate RCU-booster task that monitors per-CPU arrays of lists of target tasks that have been blocked while in an RCU read-side critical section. Overhead is incurred only when such blocking occurs, permitting the RCU read-side-acquisition primitive (e.g., `rcu_read_lock()`) to contain exactly the same sequence of instructions contained in its non-boosted counterpart.

4.1 Data Structures

Each element of each per-CPU array is a structure named `struct rcu_boost_dat` as shown in Figure 17. The `rbs_mutex` field guards all fields in the structure, `rbs_toboost` is a list containing target tasks that are candidates for boosting, and `rbs_boosted` is a list containing target tasks that have already been boosted. The next three fields govern interactions between the RCU-booster task and an exiting target task: (1) `rbs_target_wq` is a wait queue for exiting target tasks, which ensures that the RCU-booster task has finished accessing the target before the target exits, (2) `rbs_booster_wq` is a wait queue for the RCU-booster task, which ensures that any exiting target task has completed blocking before the RCU-booster task reuses the `rbs_target_wq` field, and (3) `rbs_exit_done` is the flag that the RCU-booster conditions its `wait_event()` call on. The rest of the fields are statistics: `rbs_blocked` counts the number of RCU read-side critical sections that were blocked (whether once or multiple times), `rbs_boost_attempt` counts the number of tasks that the RCU-booster task has attempted to boost, `rbs_boost_wrongstate` counts the number of such attempts that were abandoned due to the target task being in the wrong state, `rbs_boost_cmpxchgfail` counts the number of such attempts that were abandoned due to concurrent state manipulation by some other task, `rbs_boost_start` counts the number of such attempts in which boosting was started, `rbs_boost_end` counts the number of such attempts in which boosting was completed normally, `rbs_unlock` counts the number of outermost `rcu_read_unlock()` operations that end a blocked RCU read-side critical section, and `rbs_unboosted` counts the number of tasks whose boosting had to be backed out due to a concurrent `rcu_read_unlock()`. The `rbs_stats[]` array tracks the number of transitions due to each event from each state. However, these structures are organized as described in Section 3.

This approach uses the same task-struct fields as that of Section 3, shown in Figure 2, with the addition of a pointer to a `struct rcu_boost_dat` named `rcub_rbdp_wq`, which is used to mediate `exit()` processing.

```
 1 struct rcu_boost_dat {
 2   raw_spinlock_t rbs_mutex;
 3   struct list_head rbs_toboost;
 4   struct list_head rbs_boosted;
 5   wait_queue_head_t rbs_target_wq;
 6   wait_queue_head_t rbs_booster_wq;
 7   int rbs_exit_done;
 8   long rbs_blocked;
 9   long rbs_boost_attempt;
10   long rbs_boost_wrongstate;
11   long rbs_boost_cmpxchgfail;
12   long rbs_boost_start;
13   long rbs_boost_end;
14   long rbs_unboosted;
15   long rbs_unlock;
16   long rbs_stats[][];
17 }
```

Figure 17: RCU-Boost Data Structure (Complex)

Note that boosting might well be attempted concurrently with the target task removing itself from the list. Much care is required in order to avoid boosting a target task just after it removes itself from its list. Failure to avoid this scenario could result in an otherwise low-priority task remaining boosted indefinitely, in turn causing other high-priority realtime tasks to miss their deadlines. Worse yet, a task being boosted could call `exit()` immediately after exiting its critical section, possibly resulting in memory corruption due to the RCU-booster task attempting to unboost it after it had been completely removed from the system.

The state machine described in the next section prevents these failure scenarios from occurring.

4.2 State Diagram

Each task has an associated RCU-booster state, which can take on the values shown in Figure 18. Tasks in any of the yellow states are linked into the uppermost of the two sets of lists shown in Figure 3, tasks in any of the red states are linked into the lower of these two sets of lists, and tasks in any of the unshaded states are not linked into either set of lists. The black state transitions are traversed by the task, while the red transitions are traversed by the RCU-booster task. The double-walled states may be exited either of these two, requiring that the state value be atomically manipulated. Fortunately, the

RCU-booster task can only make three consecutive changes to the state before reaching a state that can only be exited by the task itself. Therefore, the number of consecutive compare-and-exchange failures for the task is limited to three, permitting $O(1)$ task-side state change.[1]

When a task resides in any of the single-walled states, however, the state may be manipulated non-atomically by the sole task permitted to exit that state.

Similarly, priority manipulations in a state that can be exited by the RCU-booster task are carried out by the RCU-booster task, even if that state can also be exited by the target task – concurrent manipulation of priorities does not reduce the number of states, but does increase the probability of bugs. However, priority manipulations in a state that can only be exited by the target task are carried out by the target task.

4.3 Per-State Details

The purpose of the individual states are as follows:

- `RCU_BOOST_IDLE`: This is the initial state. A target task resides in this state when not in an RCU read-side critical section, or when in an RCU read-side critical section that has not yet blocked.

- `RCU_BOOST_BLOCKED`: The target task has blocked while in an RCU read-side critical section.

- `RCU_BOOSTING`: The RCU-booster task has begun boosting the target task's priority.

- `RCU_BOOSTED`: The RCU-booster task has completed boosting the target task's priority.

- `RCU_UNBOOST_IDLE`: The target task exited its RCU read-side critical section while in the process of being boosted. This task has removed itself from its list, but it is the responsibility of the RCU-booster task to back out of any boosting that has taken place.

- `RCU_UNBOOST_BLOCKED`: The target task not only exited its RCU read-side critical section, but entered another one and further was blocked before the RCU-booster task managed to finish boosting. The target task will have added itself back to the appropriate list, which might well be a different one than it was one before. This situation will need to be addressed if there are ever per-CPU RCU-booster tasks.

- `RCU_UNBOOST_EXITING`: The target task not only exited its RCU read-side critical section, but also managed to invoke `exit()` before the RCU-booster task managed to finish boosting. To avoid memory corruption, the target task must wait for the RCU-booster task to get done manipulating it before completing the *exit()* code path.

- `RCU_EXIT_OK`: The RCU-booster task has finished manipulating the target task, which may therefore safely complete the *exit()* code path. This is the final state for each target task.

4.4 Events

The important events for a target task are (1) being blocked in an RCU read-side critical section, (2) completing a previously blocked RCU read-side critical section, and (3) invoking *exit()*. When a target task is blocked in an RCU read-side critical section, it always adds itself to the current list of its per-CPU array. Conversely, when a target task completes a previously blocked RCU read-side critical section, it removes itself from this list. If priority boosting has completed, it also unboosts itself. During *exit()* processing, the target task must wait for the RCU-booster task to complete any concurrent boost/unboost actions.

The important events for the RCU-booster task are (1) starting to boost a target task's priority, (2) finishing boosting a target task's priority, and (3) unboosting a target task.

[1] The locking design used in the actual implementation further limits the number of consecutive compare-and-exchange failures to one.

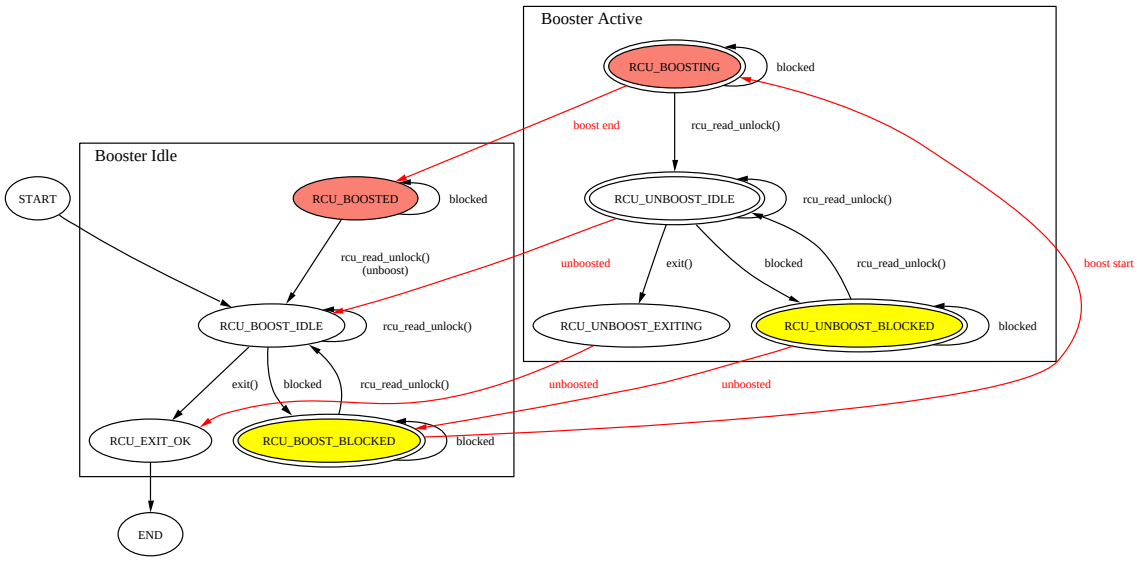

Figure 18: RCU Priority-Boosting State Diagram (Complex)

5 RCU Priority Boosting With Per-Task Lock

One of the issues with the approach described in the previous section is that RCU boosting must be explicitly accounted for in all task-priority calculations. The approach described in this section avoids this by providing a per-task lock whose main purpose is to provide priority-boosting to the target tasks, and also has a somewhat simpler state machine. This leads to a corresponding drawback, namely, that processes blocking in RCU read-side critical sections incur a latency penalty corresponding to the overhead of acquiring this additional lock.

Another drawback that became apparent only after extensive experimentation is that this method cannot work completely reliably. It is possible for the target task to block waiting for its own lock in the (unlikely, but possible) case where the RCU-booster task has just boosted it, but not yet released the lock. In this case, the target task will block waiting for the lock, and, if it is sufficiently low priority, never get a chance to run again. Since it does not yet hold the lock, the RCU-booster task is unable to priority boost it.

You should therefore instead use the approach described in Section 4.

Nevertheless, this approach is described in some detail in the hope that someone will figure out a way to make it work. After all, it is a bit simpler. Too bad about it not working in all cases!

5.1 Data Structures

The data structures are very similar to those called out in Section 4.1.

5.2 State Diagram

Figure 19 shows the state diagram for the per-task-lock version of RCU priority boosting. This fairly closely resembles the one shown in the previous section, but is slightly simpler, having one fewer node and fewer edges.

Target tasks in any of the yellow states are linked into the uppermost of the two sets of lists shown

in Figure 3, target tasks in any of the red states are linked into the lower of these two sets of lists, and target tasks in any of the unshaded states are not linked into either set of lists. Target tasks in shaded states hold the per-task lock, and the RCU-booster task might or might not hold a target task's per-task lock while the target task is in any of the RCU_END_BOOST_IDLE, RCU_END_BOOST_BLOCKED, or RCU_END_BOOST_EXITING states. The black state transitions are traversed by the task, while the red transitions are traversed by the RCU-booster task. The double-walled states may be exited either of these two, requiring that the state value be atomically manipulated.

5.3 Per-State Details

The purpose of the individual states are as follows:

- RCU_BOOST_IDLE: This is the initial state. A target task resides in this state when not in an RCU read-side critical section, or when in an RCU read-side critical section that has not yet blocked. The task does *not* hold the per-task lock.

- RCU_BOOST_BLOCKED: The target task has blocked while in an RCU read-side critical section, and holds its per-task lock. The target task will release its per-task lock upon exiting this state, so the RCU-booster task should never see this state. Of course, this state is the Achilles's heel of this approach. The target task might block on the lock, and never get a chance to run again. There are a number of amusing (but broken) strategies one might use, including reader-writer locks that the target task attempts to acquire recursively and the like.

- RCU_BOOSTING: The RCU-booster task has begun boosting the target task's priority. It first changes state, and only then acquires the target task's per-task lock. The target task can see this state either before or after the RCU-booster task has blocked on its per-task lock, but either way removes itself from the list and changes state to RCU_END_BOOST_IDLE – and only then releases its per-task lock.

- RCU_END_BOOST_IDLE: The target task completed its RCU read-side critical section after (or while in the process of) being boosted. Once the RCU-booster task acquires the lock, it will transition the state to RCU_BOOST_IDLE, then release the lock.

- RCU_END_BOOST_BLOCKED: The target task not only exited its RCU read-side critical section, but entered another one and further was blocked before the RCU-booster task managed to acquire the target task's lock. The target task will have added itself back to the appropriate list, and will hold its own per-task lock. Similarly to RCU_BOOST_BLOCKED, the target task will change state to RCU_END_BOOST_IDLE and only then release its per-task lock.

- RCU_UNBOOST_EXITING: The target task not only exited its RCU read-side critical section, but also managed to invoke `exit()` before the RCU-booster task acquired the target task's per-task lock. To avoid memory corruption, the target task must wait for the RCU-booster task to get done manipulating it before completing the *exit()* code path.

- RCU_EXIT_OK: The RCU-booster task has finished manipulating the target task, which may therefore safely complete the *exit()* code path. This is the final state for each target task.

5.4 Events

The important events for a target task are (1) blocking in an RCU read-side critical section, (2) completing a previously blocked RCU read-side critical section, and (3) invoking *exit()*. When a target task is blocked in an RCU read-side critical section, it always adds itself to the current list of its per-CPU array. Conversely, when a target task completes a previously blocked RCU read-side critical section, it removes itself from this list. If priority boosting has completed, it also unboosts itself. During *exit()* processing, the

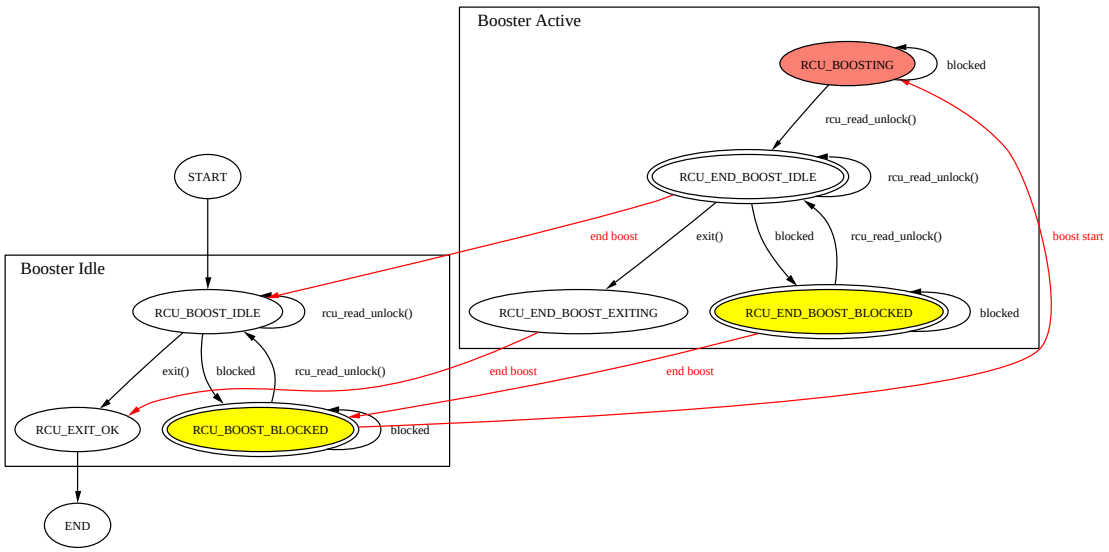

Figure 19: RCU Lock-Based Priority-Boosting State Diagram

target task must wait for the RCU-booster task to complete any concurrent boost/unboost actions.

The important events for the RCU-booster task are (1) starting to boost a target task's priority and (2) finishing boosting a target task's priority.

5.5 Important Safety Tip

Again, this lock-per-task approach simply does not work reliably in all cases. It is included only because I thought it would work, and because I feel that the cautionary tale might be valuable.

You should instead use the approach described in Section 4.

6 Issues and Roads Not Taken

Priority-boosting RCU read-side critical sections, although simple in concept, is fraught with subtle pitfalls. This section records some of the pitfalls that I fell into, in the hope that it saves others the time and trouble.

1. Aggregating lists over multiple CPUs. This might be needed for very large systems, but wait until needed before increasing the complexity. However, on large systems, it probably makes more sense to instead provide multiple RCU-boost tasks, perhaps one per CPU or per NUMA node.

2. Dedicating a per-task lock to priority-boosting of target tasks. As was noted in Section 5, this simply does not work in all cases.

3. Simpler list locking, for example, single lock per CPU. This was rejected because it could in greater contention between the RCU-booster task and target tasks. With the current design under normal conditions, contention should only occur during `rcu_read_unlock()` and `exit()` processing. Even then, multi-jiffy RCU read-side critical sections are required for the `rcu_read_unlock()` to contend with the RCU-booster task. For example, if such critical sections are blocked!

4. Immediate boosting upon blocking was rejected

due to the additional scheduling latency it imposes. (Although earlier prototypes did just this, a *very* few of them reliably so.)

5. Use of `preempt_disable()` instead of `local_irq_disable()` works, but subjects the code to extra preemption checks upon `preempt_enable()`.

6. Zero scheduling-path latency. The idea here is to require that the RCU-booster task walk the entire task list in search of tasks in need of boosting. This may be necessary for some workloads, but the overhead of the full task-list walk seems prohibitive for any server-class workload featuring large numbers of tasks.

7 Lessons Learned Along the Way

1. Interactions with RCU read-side critical sections are very touchy. By definition, a CPU can exit such a critical section with very few ripples, after all, the whole point of RCU's read-side primitives is to be very lightweight. Therefore, synchronizing with these primitives, as the RCU-booster task must, is fraught with peril. It is not that the final solution is all that complex or difficult, it is rather that there is a very large number of seductive but incorrect "solutions".

2. Interacting with tasks (which might exit at any time) can also be quite touchy. Most of the methods used to keep a task pinned down simply keep the task structure itself pinned down. Unfortunately, the RCU-booster task needs the target task to be alive and able to respond, for which a separate mechanism was (perhaps redundantly) constructed.

3. The `__rcu_init()` function is called extremely early. Not that this was a surprise, but what *was* a surprise was just how many common kernel APIs cannot be called that early. The solution is straightforward (add a second initialization function that is called later from `do_basic_setup()`, though there may well be a better solution), but this nevertheless somehow managed to be a surprise. The `system_state` variable is very helpful in marking when the scheduler becomes usable.

4. The act of designing enterprise-level "torture tests" can have the beneficial side-effect of inspiring much simpler (and thus easier to test) solutions.

5. The act of documenting a tricky algorithm can also have the beneficial side-effect of inspiring much simpler (and thus easier to document) solutions.

I had of course encountered the last two lessons much earlier in my career, but this problem offered a much-needed refresher course.

Acknowledgements

I owe a debt of gratitude to the developers and maintainers of the graphviz package, which greatly eased design of RCU priority boosting. I also owe thanks to Josh Triplett for introducing me to this invaluable package.

Legal Statement