# RCU CONTENTION FOR vCPU WITH STEAL TIME

SYS_WRITE()

↳ VFS_WRITE() (__VFS WRITE)

STRUCT *FILE → F_OP → $\boxed{\text{WRITE()}}$

↳ KERNFS_FOP_WRITE() → FILE , USER-BUF, COUNT, PPOS

↳ PRIVATE DATA = KERNFS_OPEN_FILE

↳ PRIVATE

- COPY FROM USER()

OPS = KERNFS_OPS (OF → KN)

OF → KERNFS OPEN FILE

KN → KERN FS NODE

- OPS → WRITE (OF, BUF, LEN, PPOS)

" CGROUP_FILE_WRITE ()

CGROUP CGRP = OF → KN → $\boxed{\text{PARENT}}$ → PRIV

CFTYPE CFT = OF → KN → PRIV

CGROUP_SUBSYS_STATE CSS

CFTYPE

OF → KN → PRIV

"TASKS"

· SEQ_START

· SEQ_NEXT

· SEQ_STOP

...

$\boxed{\text{, WRITE CGROUP\_TASKS\_WRITE()}}$

↳ __CGROUPS_PROCS_WRITE ()

...

PORQUE O ARQUIVO É "TASKS"

RCU/SYNC

GP_OPS: RCU_SYNC, RCU_SCHED_SYNC, RCU_BH_SYNC

1. - SYNCHRONIZE_SCHED *
2. - CALL_RCU_SCHED
3. - RCU_BARRIER_SCHED

RCU_SYNCHRONIZE:
- RCU_HEAD HEAD
- COMPLETION COMPLETION

PERCPU_DOWN_WRITE (CGROUP THREAD GROUP RWSEM) ARRAY

SCHEDULE

```
↳ RCU_SYNC_ENTER
    ↳ SYNCHRONIZE_SCHED *
        ↳ WAIT_RCU_GP (CALL_RCU_SCHED)
            ↳ __WAIT_RCU_GP (FALSE, 1, CALL_RCU_SCHED,
                  RCU_SYNCHRONIZE )
```

WAKEME AFTER RCU

CALL RCU_SCHED

WAIT COMPLETION ↳ BLOCKED HERE

INIT AND REGISTER CALLBACKS

- INIT_RCU_HEAD_ON_STACK ( ARRAY. HEAD)
- INIT_COMPLETION ( ARRAY. COMPLETION) → COMPLETION. WAIT

SEARCH [ - CALL_RCU_SCHED ( ARRAY. HEAD, WAKEME_AFTER_RCU)
↳ COMPLETES ARRAY

```
* ↳ __CALL_RCU (HEAD, FUNC, ...)
      ↳ __CALL_RCU_CORE()
```

SCHEDULA CALLBACK

LOOP WAITING FOR ARRAY COMPLETION

WAIT_FOR_COMPLETION (ARRAY. COMP

↳ CAN CALL RAISESOFTIRQ ?! *

RAISE

ou TICK_SCHED_TIMER()
TICK_NOHZ_HANDLER()

CLOCK TICK

```
└ UPDATE_PROCESS_TIMES()
    └ RCU_CHECK_CALLBACKS()   AQUI
        └ INVOKE_RCU_CORE()
            └ RAISE_SOFTIRQ (RCU_SOFTIRQ)
```

KSOFTIRQ

HANDLE

```
SOFTIRQ "RCU_SOFTIRQ"
  └ RCU_PROCESS_CALLBACKS
      └ __RCU_PROCESS_CALLBACKS()
          | CHECKS GRACE PERIOD
          | CHECKS CPU MASK FOR CALLBACKS
          └ INVOKE_RCU_CALLBACKS()
              | NO BOOST? - RCU_DO_BATCH()
              | ELSE: INVOKE_RCU_CALLBACKS_KTHREAD()
                          └ __RCU_RECLAIM
```

WAKE ME AFTER RCU

→ DÁ PRA SABER O TAMANHO DA QUEUE DE CALLBACKS? ELES FORAM EXECUTADOS POR SOFTIRQ()?

# Verification of the Tree-Based Hierarchical Read-Copy Update in the Linux Kernel

Lihao Liang
University of Oxford
lihao.liang@cs.ox.ac.uk

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

Daniel Kroening
University of Oxford
daniel.kroening@cs.ox.ac.uk

Tom Melham
University of Oxford
tom.melham@cs.ox.ac.uk

## Abstract

Read-Copy Update (RCU) is a scalable, high-performance Linux-kernel synchronization mechanism that runs low-overhead readers concurrently with updaters. Production-quality RCU implementations for multi-core systems are decidedly non-trivial. Giving the ubiquity of Linux, a rare "million-year" bug can occur several times per day across the installed base. Stringent validation of RCU's complex behaviors is thus critically important. Exhaustive testing is infeasible due to the exponential number of possible executions, which suggests use of formal verification.

Previous verification efforts on RCU either focus on simple implementations or use modeling languages, the latter requiring error-prone manual translation that must be repeated frequently due to regular changes in the Linux kernel's RCU implementation. In this paper, we first describe the implementation of Tree RCU in the Linux kernel. We then discuss how to construct a model directly from Tree RCU's source code in C, and use the CBMC model checker to verify its safety and liveness properties. To our best knowledge, this is the first verification of a significant part of RCU's source code, and is an important step towards integration of formal verification into the Linux kernel's regression test suite.

***Categories and Subject Descriptors*** [*D.2.4*]: Software/Program Verification—Model checking; [*D.1.3*]: Concurrent Programming—Parallel programming

***Keywords*** Software Verification, Parallel Computing, Read-Copy Update, Linux Kernel

## 1. Introduction

The Linux operating system kernel [1] is widely used in a variety of computing platforms, including servers, safety-critical embedded systems, household appliances, and mobile devices such as smartphones. Over the past 25 years, many technologies have been added to the Linux kernel, one example being Read-Copy Update (RCU) [18].

RCU is a synchronization mechanism that can be used to replace reader-writer locks in read-mostly scenarios. It allows low-overhead readers to run concurrently with updaters. Production-quality RCU implementations for multi-core systems must provide excellent scalability, high throughput, low latency, modest memory footprint, excellent energy efficiency, and reliable response to CPU hotplug operations. The implementation must therefore avoid cache misses, lock contention, frequent updates to shared variables, and excessive use of atomic read-modify-write and memory-barrier instructions. Finally, the implementation must cope with the extremely diverse workloads and platforms of Linux [19].

RCU is now widely used in the Linux-kernel networking, device-driver, and file-storage subsystems [19, 20]. To date, there are at least 75 million Linux servers [2] and 1.4 billion Android devices [6], which means that a "million-year" bug can occur several times per day across the installed base. Stringent validation of RCU's complex implementation is thus critically important.

Most validation efforts for concurrent software rely on testing, but unfortunately there is no cost-effective test strategy that can cover all corner cases. Worse still, some of errors that testing does detect might be difficult to reproduce, diagnose, and repair. The concurrent nature of RCU and the sheer size of the search space suggest use of formal verification, particularly model checking [4].

Formal verification has already been applied to some aspects of RCU design, including Tiny RCU [17], userspace RCU [10], sysidle [17], and interactions between dyntick-idle and non-maskable interrupts (NMIs) [16]. But these efforts either validate trivial single-CPU RCU implementations in C (Tiny RCU), or use special-purpose languages such as Promela [13]. Although special-purpose modeling languages do have advantages, a major disadvantage in the context of the Linux kernel is the difficult and error-prone translation from source code. Other researchers have applied manual proofs in formal logics to simple RCU implementations [11, 21]. These proofs are quite admirable, but require even more manual work, in addition to the translation effort.

Worse yet, Linux kernel releases are only about 60 days apart, and RCU changes with each release. Thus, any manual work must be replicated about six times a year so that mechanical and manual models or proofs remain synchronized with the Linux-kernel RCU implementation. Therefore, if formal verification is to be part of Linux-kernel RCU's regression suite, the verification methods must be scalable and automated. To this end, this paper describes how to build a model directly from the Linux kernel source code, and use the C Bounded Model Checker (CBMC) [7] to verify RCU's safety and liveness properties. To the best of our knowledge, this is the first automatic verification of a significant part of the Linux-kernel RCU source code.

## 2. Background

### 2.1 What is RCU?

Read-copy update (RCU) is a synchronization mechanism that is often used to replace reader-writer locking. RCU readers run concurrently with updaters, and so RCU avoids read-side contention by maintaining multiple versions of objects and ensuring they are not freed until all pre-existing readers complete, that is, until after a *grace period* elapses. The basic idea is to split updates into removal and reclamation phases that are separated by a grace period [18]. The removal phase removes reader-accessible references to objects, perhaps by replacing them with new versions.

Modern CPUs guarantee that writes to single aligned pointers are atomic, so that readers see either the old or new version of the data structure. These atomic-write semantics enable atomic insertions, deletions, and replacements in a linked structure. This in turn enables readers to dispense with expensive atomic operations, memory barriers, and cache misses. In fact, in the most aggressive configurations of Linux-kernel RCU, readers can use exactly the same sequence of instructions that would be used in a single-threaded implementation, providing RCU readers with excellent performance and scalability.

As illustrated in Figure 1, grace periods are only needed for those readers whose runtime overlaps the removal phase. Those that start after the removal phase cannot hold references to the removed objects and thus cannot be disrupted by objects being freed during the reclamation phase.

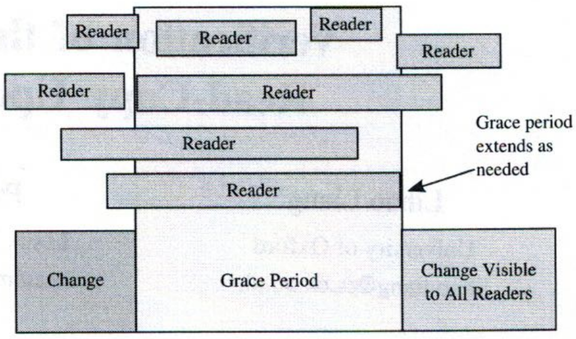

Figure 1: RCU Concepts

```
int x = 0;
int y = 0;
int r1, r2;

void rcu_reader(void) {
  rcu_read_lock();
  r1 = x;
  r2 = y;
  rcu_read_unlock();
}

void rcu_updater(void) {
  x = 1;
  synchronize_rcu();
  y = 1;
}

...

// after both rcu_reader()
// and rcu_updater() return
assert(r2 == 0 || r1 == 1);
```

Figure 2: Verifying RCU Grace Periods

### 2.2 Core RCU API Usage

The core RCU API is quite small and consists of only five primitives: `rcu_read_lock()`, `rcu_read_unlock()`, `synchronize_rcu()`, `rcu_assign_pointer()`, and `rcu_dereference()` [19].

An RCU read-side critical section begins with `rcu_read_lock()` and ends with a corresponding `rcu_read_unlock()`. When nested, they are flattened into one large critical section. Within a critical section, it is illegal to block, but preemption is legal in a preemptible kernel. RCU-protected data accessed by a read-side critical section will not be reclaimed until after that critical section completes.

The function `synchronize_rcu()` marks the end of the updater code and the beginning of the reclaimer code. It blocks until all pre-existing RCU read-side critical sections have completed. Note that `synchronize_rcu()` does not necessarily wait for critical sections that begin after it does.

Consider the example given in Figure 2. If the RCU read-side critical section in function `rcu_reader()` begins before `synchronize_rcu()` in `rcu_updater()` is called, then it must finish before `synchronize_rcu()` returns, so that the value of `r2` must be 0. If it ends after `synchronize_rcu()` returns, then the value of `r1` must be 1.

Finally, to assign a new value to an RCU-protected pointer, RCU updaters use `rcu_assign_pointer()`, which returns the new value. RCU readers can use `rcu_dereference()` to fetch an RCU-protected pointer, which can then be safely dereferenced. The returned value is only valid within the enclosing RCU read-side critical section. The `rcu_assign_pointer()` and `rcu_dereference()` functions work together to ensure that if a given reader dereferences an RCU-protected pointer to a just-inserted object, the dereference operation will return valid data rather than pre-initialization garbage.

## 3. Implementation of Tree RCU

OLD RCU MODEL

The primary advantage of RCU is that it is able to wait for an arbitrarily large number of readers to finish without keeping track every single one of them. The number of readers can be large (up to the number of CPUs in non-preemptible implementations and up to the number of tasks in preemptible implementations). Although RCU's read-side primitives enjoy excellent performance and scalability, update-side primitives must defer the reclamation phase till all pre-existing readers have completed, either by blocking or by registering a callback that is invoked after a grace period. The performance and scalability of RCU relies on efficient mechanisms to detect when a grace period has completed. For example, a simplistic RCU implementation might require each CPU to acquire a global lock during each grace period, but this would severely limit performance and scalability. Such an implementation would be quite unlikely to scale beyond a few hundred CPUs. This is woefully insufficient because Linux runs on systems with thousands of CPUs. This has motivated the creation of Tree RCU.

### 3.1 Overview

We focus on the "vanilla" RCU API in a non-preemptible build of the Linux kernel, specifically on the `rcu_read_lock()`, `rcu_read_unlock()`, and `synchronize_rcu()` primitives. The key idea is that RCU read-side primitives are confined to kernel code and, in non-preemptible implementations, do not block. Thus, when a CPU is blocking, in the idle loop, or running in user mode, all RCU read-side critical sections that were previously running on that CPU must have finished. Each of these states is therefore called a *quiescent state*. After each CPU has passed through a quiescent state, the corresponding RCU grace period ends. The key challenge is to determine when all necessary quiescent states have completed for a given grace period—and to do so with excellent performance and scalability.

For example, if RCU used a single data structure to record each CPU's quiescent states, the result would be extreme lock contention on large systems, in turn resulting in poor performance and abysmal scalability. Tree RCU therefore instead uses a tree hierarchy of data structures, each leaf of which records quiescent states of a single CPU and propagates the information up to the root. When the root is reached, a grace period has ended. Then the grace-period information is propagated down from the root to the leaves of the tree. Shortly after the leaf data structure of a CPU receives this information, `synchronize_rcu()` will return.

In the remainder of this section, we discuss the implementation of the non-preemptible Tree RCU in the Linux kernel version 4.3.6. We first briefly discuss the implementation of read/write-side primitives. We then explain Tree RCU's hierarchical data structure which records quiescent states while maintaining bounded lock contention. Finally, we discuss how RCU uses this data structure to detect quiescent states and grace periods without individually tracking readers.

### 3.2 Read/Write-Side Primitives

In a non-preemptible kernel, any region of kernel code that does not voluntarily block is implicitly an RCU read-side critical section. Therefore, the implementations of `rcu_read_lock()` and `rcu_read_unlock()` need do nothing at all, and in fact in production kernel builds that do not have debugging enabled, these two primitives have absolutely no effect on code generation.

In the common case where there are multiple CPUs running, the update-side primitive `synchronize_rcu()` calls `wait_rcu_gp()`, which is an internal function that uses a callback mechanism to invoke `wakeme_after_rcu()` at the end of some later grace period. As its name suggests, `wakeme_after_rcu()` function wakes up `wait_rcu_gp()`, which returns, in turn allowing `synchronize_rcu()` to return control to its caller.

### 3.3 Data Structures of Tree RCU

RCU's global state is recorded in the `rcu_state` structure, which consists of a tree of `rcu_node` structures with a child count of up to 64 (32 in a 32-bit system). Every leaf node can have at most 64 `rcu_data` structures (again 32 on a 32-bit system), each representing a single CPU, as illustrated in Figure 3. Each `rcu_data` structure records its CPU's quiescent states, and the `rcu_node` tree propagates these states up to the root, and then propagates grace-period information back down to the leaves. Quiescent-state information does not propagate upwards from a given node until a quiescent state has been reported by each CPU covered by the subtree headed by that node. This propagation scheme dramatically reduces the lock contention experienced by the upper levels of the tree. For example, consider a default `rcu_node` tree for a 4,096-CPU system, which will have have 256 leaf nodes, four internal nodes, and one root node. Dur-

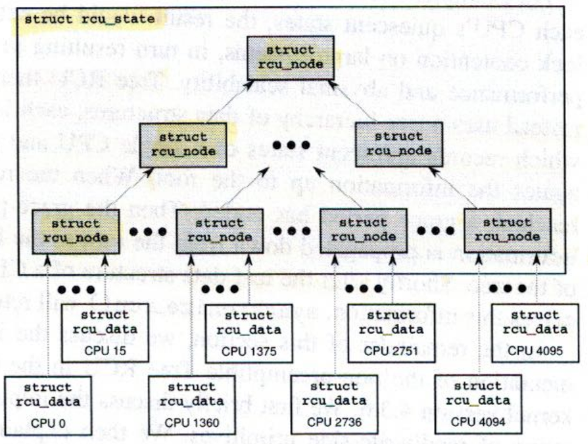

Figure 3: Tree RCU Hierarchy

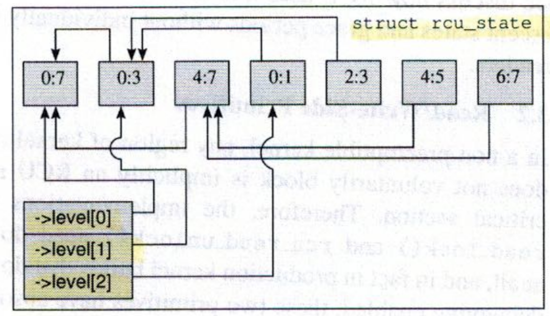

Figure 4: Array Representation for a Tree of rcu_node Structures

ing a given grace period, each CPU will report its quiescent states to its leaf node, but there will only be 16 CPUs contending for each of those 256 leaf nodes. Only 256 of the CPUs will report quiescent states to the internal nodes, with only 64 CPUs contending for each of the four internal nodes. Only four CPUs will report quiescent states to the root node, resulting in extremely low contention on the root node's lock, so that contention on any given rcu_node structure is sharply bounded even in very large configurations. The current RCU implementation in the Linux kernel supports up to a four-level tree, and thus in total $64^4 = 16,777,216$ CPUs in a 64 bit machine.[1]

### 3.3.1 rcu_state Structure

Each flavor of RCU has its own global rcu_state structure. The rcu_state structure includes a array of rcu_node structures organized as a tree struct rcu_node node[NUM_RCU_NODES], with rcu_data structures connected to the leaves. Given this organization, a breadth-first traversal is simply a linear scan of the array. Another array struct rcu_node *level[NUM_RCU_LVLS] is used to point to the left-most node at each level of the tree, as shown in Figure 4.

The rcu_state structure uses unsigned long fields ->gpnum and ->completed to track RCU's grace periods. The ->gpnum field records the most recently started grace period, whereas ->completed records the most recently ended grace period. If the two numbers are equal, then corresponding flavor of RCU is idle. If gpnum is one greater than completed, then RCU is in the middle of a grace period. All other combinations are invalid.

### 3.3.2 rcu_node Structure

The tree of rcu_node structures records and propagates quiescent-state information from the leaves to the root, and also propagates grace-period information from the root to the leaves. The rcu_node structure has a spinlock ->lock to protect its fields. The ->parent field references the parent rcu_node structure, and is NULL for the root. The ->level field indicates the level in the tree, counting from zero at the root. The ->grpmask field identifies this node's bit in the ->qsmask field of its parent. The ->grplo and ->grphi fields indicates the lowest and highest numbered CPU that are covered by this rcu_node structure, respectively.

The ->qsmask field indicates which of this node's children still need to report quiescent states for the current grace period. As with rcu_state, the rcu_node structure has ->gpnum and ->completed fields that have values identical to those of the enclosing rcu_state structure, except at the beginnings and ends of grace periods when the new values are propagated down the tree. Each of these fields can be smaller than its rcu_state counterpart by at most one.

### 3.3.3 rcu_data Structure

The rcu_data structure detects quiescent states and handles RCU callbacks for the corresponding CPU. The structure is accessed primarily from the corresponding CPU, thus avoiding synchronization overhead. As with the rcu_state structure, different flavors of RCU maintain their own per-CPU rcu_data structures. The ->cpu field identifies the corresponding CPU, the ->rsp field references the corresponding rcu_state structure, and the ->mynode field references the corresponding leaf rcu_node structure. The ->grpmask field identifies this rcu_data structure's bit in the ->qsmask field of its leaf rcu_node structure.

The rcu_data structure's ->qs_pending field indicates that RCU needs a quiescent state from the corresponding CPU, and the ->passed_quiesce indicates that the CPU has already passed through a quiescent state. The rcu_data also has ->gpnum and ->completed fields, which can lag arbitrarily behind their counterparts in the rcu_state and rcu_node structures on idle CPUs. However, on the non-idle CPUs that are the focus of this paper, they can lag at most one grace period behind their leaf rcu_node counterparts.

The rcu_state structure's ->gpnum and ->completed fields represent the most current values, and are tracked

[1] Four-level trees are only used in stress testing, but three-level trees are used in production by 4096-CPU systems.