National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Systematic Concurrency Testing of Read-Copy-Update under Sequentially Consistent and Weak Memory Models

Diploma Thesis

MICHALIS KOKOLOGIANNAKIS

Supervisor : Konstantinos Sagonas
Associate Professor NTUA

Athens, October 2016

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Systematic Concurrency Testing of Read-Copy-Update under Sequentially Consistent and Weak Memory Models

Diploma Thesis

MICHALIS KOKOLOGIANNAKIS

Supervisor : Konstantinos Sagonas
Associate Professor NTUA

Approved by the examining committee on the October 13, 2016.

....	....	....
Konstantinos Sagonas Associate Professor NTUA	Nikolaos S. Papaspyrou Associate Professor NTUA	Nectarios Koziris Professor NTUA

Athens, October 2016

....

Michalis Kokologiannakis
Electrical and Computer Engineer

Abstract

Thorough verification and testing of concurrent programs is an important, but also challenging problem. Consider, for instance, a library for synchronization between processes, or for providing access to some shared data structure by many threads. When setting up a thorough testing procedure for such a library, one must consider: the many possible ways in which client threads can access this library, all the different ways in which the threads in the library itself can interact, and any possible effects of the weak memory consistency models which are employed in modern microprocessors.

This thesis investigates the systematic concurrency testing of the Read-Copy-Update library, a synchronization mechanism used heavily in the Linux kernel. For this purpose we used Nidhugg, a stateless model checking tool for C/pthreads programs, which incorporates extensions for checking effects of weak memory models such as TSO, PSO and POWER. An informal, yet precise specification for Read-Copy-Update has been constructed, along with a suitable systematic test suite for it. In this thesis, we present the first mechanical validation of the Grace-Period guarantee of Linux kernel's Tree RCU implementation, for non-preemptible environments. Furthermore, we managed to reproduce a known bug in the kernel using Nidhugg, and to prove that a previously reported bug does not in reality qualify as a bug. Finally, some other properties of Tree RCU and Tiny RCU have been tested as well.

Key words

Formal Verification, Stateless Model Checking, Systematic Concurrency Testing, Synchronization Techniques, RCU, Read-Copy-Update, Deferred Destruction, Linux Kernel, Parallel Programming.

Acknowledgements

First of all, I am most grateful to my advisor, Kostis Sagonas, for all his help and support during this effort. With his ceaseless enthusiasm and avid interest in research he managed to motivate and inspire me, while his essential advice and guidance throughout this process have been invaluable. I absolutely enjoyed every minute of working with him for my thesis during the last year, and I am looking forward to working with him again in the future.

I am also much obliged to Paul McKenney, for dedicating a lot of his free time to answer all of my questions willingly and promptly. With his profound insight into RCU-related issues he helped me in numerous occasions, and his suggestions and input were extremely helpful.

Finally, I would like to extend my thanks to my parents and brother for their patience, love and support throughout these years; after all, they are the ones who made this endeavour possible.

Michalis Kokologiannakis,

Athens, October 13, 2016

This thesis is also available as Technical Report CSD-SW-TR-1-16, National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Software Engineering Laboratory, October 2016.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

Abstract 5

Acknowledgements 7

Contents 9

List of Figures 11

List of Listings 13

1. Introduction 15

2. Design and Architecture of Modern Linux Kernels 17
- 2.1 Memory Ordering 17
  - 2.1.1 Compiler Instruction Reordering 17
  - 2.1.2 CPU Instruction Reordering 18
- 2.2 The Scheduler 20
  - 2.2.1 Preemptive vs Non-Preemptive Scheduling 21
  - 2.2.2 Scheduler Calls 22
- 2.3 Interrupts 22
  - 2.3.1 What is an Interrupt? 22
  - 2.3.2 Handling an Interrupt 23
  - 2.3.3 I/O Interrupts 23
- 2.4 Bottom Halves and Deferrable Functions 26
  - 2.4.1 Actions in Deferrable Functions 26
  - 2.4.2 Softirqs 27
- 2.5 Time 30
  - 2.5.1 HZ 30
  - 2.5.2 Jiffies 31
  - 2.5.3 Tickless Operation 31
- 2.6 Synchronization Methods 31
  - 2.6.1 Atomic Operations 32
  - 2.6.2 Spinlocks 32
  - 2.6.3 Semaphores 33
  - 2.6.4 Mutexes 34
  - 2.6.5 Completions 35

3. Read-Copy-Update (RCU) 37
- 3.1 Introduction to RCU 37
- 3.2 RCU Fundamentals 38
  - 3.2.1 Publish–Subscribe Mechanism 38
  - 3.2.2 Waiting for Pre-Existing Readers to Complete 41
  - 3.2.3 Maintaining Multiple Versions of Recently Updated Objects 44

3.3 Conclusions Drawn from the Usage of RCU 45
3.3.1 Some Remarks 45
3.3.2 Comparison with Other Synchronization Mechanisms 46
3.4 RCU Specifications 48
3.4.1 Grace-Period Guarantee 48
3.4.2 Publish-Subscribe Guarantee 50
3.4.3 Unconditional Read-to-Write Upgrade 50
3.4.4 Unconditional Execution of the RCU Primitives 50

4. Verification of Tiny RCU 53
4.1 General Information 53
4.2 Kernel Environment Modeling 54
4.2.1 CPU, Interrupts & Scheduling 54
4.2.2 Kernel Definitions 56
4.3 Validating Tiny RCU's Grace-Period Guarantee 56

5. Verification of Tree RCU 59
5.1 General Information 59
5.1.1 High-Level Explanation 60
5.1.2 Data Structures Involved 62
5.1.3 Use Cases 65
5.2 Kernel Environment Modeling 68
5.2.1 CPU, Interrupts & Scheduling 68
5.2.2 Kernel Definitions 71
5.2.3 Synchronization Mechanisms 71
5.3 Reproducing Older Bugs 72
5.3.1 Bug #1: Synchronization issues for `rcu_process_gp_end()` 72
5.3.2 Bug #2: Bug between grace-period forcing and initialization 75
5.4 Validating Tree RCU's Publish-Subscribe Guarantee 76
5.5 Validating Tree RCU's Grace-Period Guarantee 77

6. Concluding Remarks 87

Bibliography 89

Appendix 95

A. Tree RCU Modified Functions 95

List of Figures

3.1 Deferred Deletion via RCU (adapted from [McKe13]). 38
3.2 Waiting for pre-existing RCU readers to complete (adapted from [McKe13]). 43
3.3 The reclamation phase starts after a grace period has elapsed (adapted from [McKe07c]. 44

5.1 Tree RCU node hierarchy (adapted from [McKe08a]). 60
5.2 A simple hierarchy example (adapted from [McKe08a]). 61
5.3 Tree RCU node hierarchy – full (adapted from [McKe08a]). 64

List of Listings

2.1 Simple program to demonstrate compiler instruction reordering. 18
2.2 Assembly code for Listing 2.1. 18
2.3 Optimized assembly code for Listing 2.1. 18
2.4 Publish-Subscribe example – Publisher's code. 19
2.5 Publish-Subscribe example – Subscriber's code. 19
2.6 Two-threaded program prone to Store-Load reordering. 20
2.7 Enumeration type for existing softirqs from `kernel/interrupt.h`. 27
2.8 Structure representing a single softirq entry. 28
2.9 Atomic data structures in Linux kernel. 32
2.10 Completion data structure. 36
3.1 Publish reordering effects. 39
3.2 Publish mechanism. 39
3.3 Subscribe reordering effects. 40
3.4 Subscribe reordering effects – optimized code. 40
3.5 Subscribe mechanism. 40
3.6 Typical RCU update example. 42
3.7 Definition for `list_replace_rcu()`. 45
3.8 RCU is a reference-counting mechanism. 46
3.9 RCU's Grace-Period guarantee example. 49
3.10 RCU's Grace-Period guarantee example 2. 49
3.11 RCU's Publish-Subscribe guarantee example. 51
4.1 `synchronize_sched()` implementation for Tiny RCU. 54
4.2 Fake CPU acquisition for Tiny RCU. 55
4.3 Fake CPU release for Tiny RCU. 55
4.4 `cond_resched()` function for Tiny RCU. 56
4.5 Litmus test for validation of Tiny RCU's Grace-Period guarantee. 57
5.1 Fake CPU acquisition for Tree RCU. 68
5.2 Fake CPU release for Tree RCU. 69
5.3 Unsynchronized accesses to the `->completed` counter. 73
5.4 Too-short grace period caused by synchronization issues in `rcu_process_gp_end()`. 82
5.5 Segment of `force_quiescent_state()` function in kernel v2.6.31.1. Code from file `kernel/rcutree.c`. 83
5.6 Test for validation of Tree RCU's Publish-Subscribe guarantee. 84
5.7 Litmus test for validation of Tree RCU's Grace-Period guarantee. 85

Chapter 1

Introduction

All software developers have experience with software that did not work, or did not work as expected. Software reliability can have a huge impact on the software users, with unreliable software being responsible for a variety of problems: from lost time (probably spent for debugging) to even death (e.g., if a safety-critical system fails to perform as intended). Taking this into consideration, along with the fact that a big percentage of today's software is designed to run concurrently, we come to the conclusion that the correct behaviour of concurrent programs is undoubtedly of vital importance.

However writing concurrent programs that operate as intended is much harder than writing single-threaded programs. There is a much bigger set of risks, which includes all the hazards present in single-threaded programs, plus some -usually adverse- effects one has to consider when writing concurrent software. Dangers present only in concurrent programs include (but are not limited to): race conditions, deadlocks, livelocks and data races. In addition, one has to be aware of the relaxed (weak) memory consistency models [Wikia, Rela] employed in modern microprocessors. In weak memory models it is possible to experience memory reorderings (see Section 2.1.2) which are used to fully utilize the bus-bandwidth of different types of memory, such as caches and memory banks. Consequently, if memory barriers are not effectively used, the behaviour of a concurrent program may not match the expected one.

Verification of Concurrent Programs

Verification and testing of concurrent programs is difficult. *Model checking* [Wikib] is a technique that systematically explores the state space of a given program and verifies that each reachable state meets a given specification. In other words, model checking is a method for formally verifying concurrent systems; specifications about the system are expressed as temporal logic formulas, and efficient algorithms are used to traverse the model defined by the system and check whether the specification holds.

Having said that, model checking tools face a combinatorial explosion of the state-space, since capturing and storing a large number of global states is required. Therefore, clever techniques have emerged in order to tackle this problem.

For example, bounded model checking [Bier03] unrolls the finite state machine for a fixed number of steps k, and checks for property violations within these k steps. Bounded model checking combined with the usage of partial orders for modelling program executions [Algl13] has been effectively implemented in tools like CBMC [Clar04].

Stateless model checking [Gode97], also known as systematic concurrency testing, is a technique with low memory requirements that is applicable to programs with executions of bounded length. Stateless model checking tools explore the state space of a program without explicitly storing global states. The technique has been successfully implemented in tools such as VeriSoft [Gode05], CHESS [Musu08], Concuerror [Chri13], and recently Nidhugg [Abdu15]. Some of these tools also try to combat the problem of combinatorial explosion in the number of interleavings that they examine in order to maintain full coverage of all behaviours by using *partial order reduction* [Valm91, Pele93, Gode96, Clar99] techniques. Partial order reduction is based on the observation that two interleavings can be considered equivalent if one can be obtained from the other by swapping adjacent, independent execution steps. *Dynamic Partial Order Reduction (DPOR)* techniques capture dependencies betwen operations of concurrent threads while the program is running [Flan05]. The exploration begins with an arbitrary interleaving whose steps are then used to identify dependent operations and points where alternative interleavings need to be explored in order to capture all program behaviours.

Stateless model checking can be extended to handle weak memory models. Nidhugg [Abdu15], for example, is a stateless model checker for C/C++ programs that use pthreads, which incorporates extensions for finding bugs caused weak memory models such as TSO, PSO and POWER. Nidhugg's implementation employs a very efficient dynamic partial order algorithm called source-DPOR [Abdu14]. In this thesis we used Nidhugg for all our tests.

Aim of this Thesis

This thesis intends to thoroughly test the Read-Copy-Update (RCU) mechanism used in Linux kernel [Linu], in order to validate its basic guarantees. RCU is a synchronization mechanism used heavily in Linux kernel, and many of the kernel's subsystems rely on RCU's correct operation. Of course, the fact that the Linux kernel is present in more than a billion devices [Call15] makes the testing of RCU of even greater importance.

Previous work on RCU verification includes the expression of RCU's formal semantics in terms of separation logic [Gots13] and the verification of userspace-RCU in a logic for weak memory [Tass15]. A virtual architecture to model out-of-order memory accesses and instruction scheduling has been proposed [Desn13], and a verification of userspace-RCU has been done using the SPIN model checker [Desn12]. Moreover, researchers at Stony Brook University produced an RCU-aware data-race detector [Dugg10, Seys12]. Paul E. Mckenney verified Tiny RCU (a flavour of RCU for uniprocessor systems) using CBMC [McKe15d], and this was the first attempt on verifying the kernel's code directly. Finally, mutation testing strategies have been applied to the Linux-kernel RCU [Ahme15] as well.

Our contribution includes the verification of the basic properties of Tiny RCU and Tree RCU, both being implementations used in the Linux kernel. For this, we present an informal, yet precise specification of RCU, along with a test case suite which was developed from this specification. Additionally, we managed to reproduce a known kernel bug, and to demonstrate that a previously reported bug does not in reality qualify as a bug. All of our tests use the source code from the kernel directly with only a few modifications, while most of the previous attempts on the verification of RCU used models that required manual translation of RCU's source code. In addition, four different versions of the Linux kernel were included, in order to cover a wide range of behaviours in our tests.

Brief Overview

In Chapter 2 we discuss how the compiler and the CPU may change the behaviour of a concurrent program, along with the basic subsystems of the Linux kernel on which RCU relies. In Chapter 3 the fundamental parts of RCU along with a specification for RCU are presented. In Chapters 4 and 5, we discuss the way we emulated the kernel's environment and the development of a suitable systematic test suite for Tiny RCU and Tree RCU, respectively. In Chapter 6 we summarize the previous chapters and the conclusions we drew from this thesis, and present some possible extensions to our work.

Chapter 2

Design and Architecture of Modern Linux Kernels

The kernel is the basic and most crucial component of the Linux Operating System. The kernel is the software responsible for program execution and for providing safe access at the computer's hardware. In this chapter, we will focus on some elements of the kernel which play a major role in the smooth operation of the kernel and the OS itself, and are inextricably intertwined with the implementation and the correct operation of RCU. But first, we present some key factors that play a crucial role in the behaviour of concurrent programs.

2.1 Memory Ordering

One of the factors responsible for changing the behaviour of a program, is the memory ordering in modern microprocessors.

When writing a program of any kind, it is at least desirable for this program to run fast. The whole infastructure available to the programmer, from hardware to software, is designed and optimized to satisfy this exact requirement: the minimization of a program's execution time.

However, as all things, this comes at a price. The memory interactions of the source code may be reordered, and these reorderings may happen both at the compilation stage (by the compiler) and at the execution stage (by the CPU; also called *out of order execution*). Surely, all these reorderings must abide by some rules. Surprisingly(?) enough, the fundamental rule followed by compiler developers and hardware designers, only states that the behaviour of a single-threaded program must not change. In parallel programming, though, there might be some unwanted behaviours if one is not careful, especially in *lock-free programming*.

In this section, we will examine the memory reorderings made both by the compiler and by the processor, the effects these may have, and the ways to prevent them.

2.1.1 Compiler Instruction Reordering

First, we will discuss instruction reorderings performed by the compiler. The compiler is the program responsible for transforming the source code into machine code. During this transformation, the compiler is free to perform many transformations and reorderings, as long as the program behaviour does not change. In order to demonstrate such reorderings, consider the following example:

```c
int x, y;

void foo()
{
  x = y + 42;
  y = 0;
}
```

Listing 2.1: Simple program to demonstrate compiler instruction reordering.

We can generate the assembly code for this function using the compiler option `-S`, in both the gcc and the clang compiler. For this example, we used the gcc compiler on a Debian Jessie 64-bit computer (gcc 4.9.2). We also used the option `-masm=intel`, to change the dialect of the produced assembly. The assembly code of interest, corresponding to this function's body, is presented in Listing 2.2.

```
mov   eax, DWORD PTR y[rip]
add   eax, 42
mov   DWORD PTR x[rip], eax
mov   DWORD PTR y[rip], 0
```

Listing 2.2: Assembly code for Listing 2.1.

On the other hand, if we compile the program with optimizations enabled, using the `-O2` option, the assembly we get is quite different; see Listing 2.3.

```
mov   eax, DWORD PTR y[rip]
mov   DWORD PTR y[rip], 0
add   eax, 42
mov   DWORD PTR x[rip], eax
```

Listing 2.3: Optimized assembly code for Listing 2.1.

One can see that the compiler *did* reorder the two instructions. But while this reordering may seem harmless (and indeed it is, for single-threaded programs), it may cause troubles, if one is not careful. Imagine what would happen if we had a thread with two global variables, `x` and `y` respectively, with `y` being a flag indicating whether `x` is initialized properly. Suppose that after `x`'s initialization, `y` is set to one. If another thread wishes to know if `x` is initialized properly, the only action it has to do is read the value of `y`. If the two stores were reordered, a thread may mistakely believe that `x` is initialized properly while it has not, and then all sorts of bad things may happen.

On the other hand, the clang compiler does *not* reorder these two instructions, and the above scenario will work just fine, even on multicore environments. But, one should be aware of all the possible compiler optimizations, and the nasty repercussions they may have. Fortunately for those who write lock-free code, every compiler comes with some barrier instructions that prevent this kind of optimizations (e.g. `asm volatile("" ::: "memory")` for gcc).

2.1.2 CPU Instruction Reordering

We may have briefly discussed memory orderings happening at compile time but, as already mentioned, that is only half the story. Every processor performs optimizations of its own, which, of course, result in memory accesses being reordered at runtime. Again, these optimizations *do not* affect the behaviour of single-threaded programs, but may cause trouble in multi-threaded programming, and only become apparent in multicore architectures.

In this section, we will discuss the different types of reordering a CPU is free to take. For our purposes, we can distinguish four basic types of instruction reordering:

Store-Store: The data of the second store may be visible to other processors before those of the first store.

Load-Load: The data accessed by the second load may be loaded before those of the first load.

Store-Load: The data accessed by the load instruction may be loaded before the data of the store become visible to other processors.

Load-Store: The data associated with the store may become visible to other processors before the data accessed by the load are loaded.

We will further investigate these types of reorderings in the next subsections, but keep in mind that, in all CPUs, there are memory barrier instructions to prevent them.

Store-Store reordering

The first type of reordering we will be looking into is Store-Store reordering. As its name denotes, this reordering allows a store instruction to be reordered with a subsequent store instruction. This can be harmful since, if the first store's data are not flushed to memory -and hence visible to other processors- *before* a subsequent's store data are, there is the risk of another CPU seeing the data of a subsequent store first. "The CPU is not so dumb to reorder stores if there is some kind of dependency between them", one might argue. Well, things are not so simple. Consider the following code snippet:

```
x = 42;
/* store-store barrier needed here to prevent the reordering */
is_published = 1;
```

Listing 2.4: Publish-Subscribe example – Publisher's code.

This is a classic Publish-Subscribe example, where a variable is set *after* another one is initialized. Obviously, we want the variable `is_published` to be set to one after variable `x` is initialized. However, if the value of `is_published` becomes visible to other CPUs before `x` is initialized, another CPU trying to read the value of `x` may end up reading garbage. In order to avoid these type of reordering, we need a "Store-Store" barrrier (CPU specific), something much needed when writing lock-free code designated to run in CPUs which do not guarantee strict ordering of flushes from write buffers.

Load-Load reordering

Moving on to Load-Load reordering, this type of reordering occurs when a load operation gets reordered with a subsequent load operation. To better understand the possible consequences of this, the classic Publish-Subscribe example will be examined, but from the subscriber's side this time. Consider the following code:

```
if (is_published) {
  /* load-load barrier needed here to prevent the reordering */
  return x;
}
```

Listing 2.5: Publish-Subscribe example – Subscriber's code.

In this example, if the value of `x` is loaded before the value of `is_published` a wrong value for `x` will be returned. If the loads are reordered, the value of `x` read might be older than the flag indicating that the data is published. A Load-Load fence is required to prevent this kind of reordering. These kind of fences are required on processors that perform speculation optimizations and/or out-of-order execution which supports this kind of reordering.

Store-Load reordering

The type of reordering a developer is most likely to observe is the Store-Load reordering. That is because it can occur in computers with Intel processors of the x86/64 architecture, which make the majority of CPUs out there.

A Store-Load reordering means that a store operation can be reordered with a subsequent load operation. With this type of reordering, the result `r_1 == r_2 == 0` is possible in the following code snippet which includes the assembly codes for two CPUs, running in parallel:

```
; CPU0's code
mov [x], 1
; store-load barrier needed here to prevent the reordering
mov r1, [y]

; CPU1's code
mov [y], 1
; store-load barrier needed here to prevent the reordering
mov r2, [x]
```

Listing 2.6: Two-threaded program prone to Store-Load reordering.

While this seems counter-intuitive at first sight, it is definitely possible since the data of a store may be flushed -and hence be visible to other CPUs- *after* loading the data of the subsequent load. Not surprisingly, the only way to prevent this kind of reordering is by inserting a Store-Load memory fence, which is, in general, a pretty expensive barrier.

Load-Store reordering

The last type of memory reordering is Load-Store reordering. This kind of reordering allows the data accessed by a load instruction to be loaded *after* a subsequent store instruction is flushed. Of course, for this type of reordering to take place, the CPU must perform an out-of-order execution that permits waiting store instructions to bypass loads. On a real CPU, if there is a cache miss on a load followed by a cache hit on the store, and there is no dependency between them, such reordering might indeed happen. Like all the other types of reorderings we have discussed, a Load-Store fence would prevent this reordering from happening.

2.2 The Scheduler

This section consists of a brief introduction of the Linux process scheduler, or, to put it simply, the kernel's subsystem which gives processes the ability to run [Love10].

The *process scheduler* (or just the *scheduler*) is the subsystem responsible for deciding which process will run, when, and for how long. The scheduler forms the basis of a multitasking OS like Linux, since it divides processing time to the system processes. By deciding which process will run, the scheduler is responsible for the optimal usage of the system resourses, while providing users the illusion of concurrent program execution. The processes which are waiting to run are called *runnable*.

2.2.1 Preemptive vs Non-Preemptive Scheduling

A *multitasking* operating system is an operating system which is capable of simultaneously interleaving the execution of more than one process. While this gives the impression of concurrent program execution even on uniprocessor machines, it also allows parallel process execution in different processing units on a multicore system. A multitasking OS allows processes to block or sleep, hence, to not execute if there is not any work to be done. These processes are not runnable, even though they reside in memory, because they are probably waiting for an event (e.g., input from the keyboard). Consequently, only one process will be in runnable state per CPU, even though many may reside in memory.

Considering the aforementioned, it becomes conspicuous that multitasking relies on the scheduler. There are two ways of multitasking and, consequently, sheduling:

Non-preemptive multitasking: Also called *cooperative multitasking*. In this type of multitasking, a process can stop running only voluntarily, i.e. there is no way of forcing a process out of the CPU. The act of a process voluntarily suspending itself is called *yielding*.

Preemptive multitasking: In preemptive multitasking it is the scheduler's decision when a process will stop running, and a new one starts to run. The act of involuntarily suspending a running process is called *preemption*.

Linux, just like most Unix variants and the majority of modern operating systems, implements preemptive multitasking. In preemptive multitasking, the time for which a process runs before it gets preempted from the scheduler is usually predetermined, and it is called the process' *timeslice*. Essentially, the timeslice gives every process a slice of the processor's time. The scheduler is thus able to take global scheduling decisions regarding the system, just by managing the timeslice. This can be done either statically or dynamically. Of course, in modern operating systems the timeslice for a process is calculated dynamically, according to the process behaviour and some configurable system policies (e.g., `NICE` values).

Conversely, in cooperative multitasking, processes stop running voluntarily. Naturally, it is desirable that the system resources are used in an optimal manner. Therefore, frequent yielding on behalf of the processes seems desirable, since in this way every process will have a chance to run, and a portion of the processor. Of course, the operating system cannot enforce this kind of behaviour. An inherent repercussion of this event is the ability of a process to run for as long as it desires, monopolizing the processor and, possibly, freezing the entire system. As can be seen, in this case, the presentation of ghastly events is the best way to emphasize the scheduler's inability to control the timeslice of a process.

Nowadays, most operating systems use preemptive multitasking. Some notable, relatively recent, examples of systems that used cooperative multitasking are Windows before Windows 95 and Windows NT, and the Classic Mac OS. In Linux, the kernel is tunable, in the sense that it gives the user the ability to choose between a preemptive and a non-preemptive version. The preemptive version's scheduler is called Completely Fair Scheduler (CFS), although the explanation of CFS's operation is beyond the scope of this thesis.

2.2.2 Scheduler Calls

Another point of interest, strictly correlated with the scheduler's operation, are the points where the scheduler is called (implicitly or explicitly), and the functions that perform these calls.

In order for the scheduler to know when a process needs to be preempted, a flag is used, namely the `TIF_NEED_RESCHED` flag. By checking this flag's value, the scheduler is able to determine if a process needs to be preempted. The points at which the scheduler may be called are (see `kernel/sched/core.c`, function `__schedule()`):

- In a non-preemptive kernel:
  - Anywhere `cond_resched()` is called.
  - Anywhere `schedule()` is called.
  - When returning from a syscall or an exception to user space.
  - When returning from an interrupt handler to user space.
- In a preemptive kernel:
  - In syscall or exception context, or at the outmost call of `preempt_enable()`.
  - In interrupt context, when returning to preemptible context from an interrupt handler (e.g., in timer interrupts, if the process' timeslice has ended and the process gets preempted, etc).

Lastly, regardless of the kernel type, when a process blocks, `__schedule()` function may be called, which directly invokes the scheduler. A process might block because of a semaphore, a mutex, a wait queue, etc.

2.3 Interrupts

In the previous section, we examined the kernel's subsystem which is responsible for process interleaving, the scheduler. Although we only scratched the surface of a very complex subsystem, understanding its purpose and the way it works is absolutely necessary if we want to fully understand the way RCU works. In this section, we will be dealing with interrupts, and the kernel's subsystem which is responsible for handling them.

2.3.1 What is an Interrupt?

First of all, what is an *interrupt*? An *interrupt request*, or IRQ, is a request for a service sent at hardware level, raised by hardware of software, and indicating an event that needs the CPU's immediate attention [Bove05]. An interrupt alerts the processor that a high priority event has occured, that requires the *interruption* of the current code the processor is executing – hence its name. Interrupts may be sent through a dedicated hardware line (at hardware level), or across a hardware bus as an information packet (called Message Signaled Interrupt, or MSI).

An interrupt may occur in plenty of cases, for example, when a user presses a button on the keyboard. When an interrupt is emitted, the processor responds by suspending its activities, saving its state, and calling a special function called an *interrupt handler* to deal with the interrupt. When this function finishes processing the interrupt, the processor resumes its normal activities.

2.3.2 Handling an Interrupt

So, how does the Linux kernel handle interrupts? In general, interrupt handling depends on the *type* of the interrupt. We can distinguish three main types of interrupts [Bove05]:

Input/Output Interrupts (I/O Interrupts): An I/O device requires the CPU's attention. The corresponding interrupt handler has to query the device to determine the proper course of action.

Timer Interrupts: Some timer has issued an interrupt. This may be either a local Advanced Programmable Interrupt Controller (APIC) timer, or an external timer. This type of interrupt occurs to inform the processor that a specific, predefined, architecture-dependent time interval has occured[1]. These interrupts are handled mostly like I/O Interrupts. Of course, there are some differences (related to timekeeping), but the course of actions in these two types is very much alike.

Interprocessor Interrupts (IPIs): A processor has issued an interrupt to another processor of a multicore system. Interprocessor Interrupts can be used for interrupt-driven communication between the CPUs of a multicore system (SMP).

We will come back to these categories, and specially the first one, in a bit.

In Linux, interrupt handlers are normal C functions. They do conform to some very specific guidelines —as pretty much everything in the kernel does— but there is nothing too peculiar with them. The context in which interrupts run is called *interrupt context*, and we will discuss it later in this chapter.

Since interrupts... well, interrupt the execution of the current code the CPU is running, interrupt handlers have to be *quick*. This is the main property one has to keep in mind when writing code for a device driver[2]. Of course, there may be a substantially big amount of work to be done when handling an interrupt, and that surely can prevent the interrupt handler from being quick. That is why the processing of interrupts is cut in half. The interrupt handler is the *top half*, and performs only time-critical work, while work that can be performed later is deffered until the *bottom half*. Bottom halves and the mechanisms that are used to implement them are discussed in Section 2.4.

Lastly, while the duties of an interrupt handler vary, the bare minimum a handler has to do is acknowledge the interrupt to the APIC. This is the most essential and crucial work an interrupt handler has to perform. The rest of the work it has to perform depends on the type of the interrupt and the handler itself.

2.3.3 I/O Interrupts

This section consists of a more in-depth description of the first type of interrupts, the I/O Interrupts. We will discuss how these interrupts are handled as well as which actions need to be performed when an interrupt of this type occurs.

First of all, an I/O interrupt handler needs to be flexible enough in order to service several devices at the same time. In some architectures, for example the PCI bus architecture, several devices may share the same IRQ line, which makes the process of distinguishing the device that issued the interrupt a bit difficult. However, some hardware devices used mostly in computers with older architectures (e.g., Industry Standard Architecture, or ISA) do not operate reliably, if they share the same IRQ line with another hardware device.

We can achieve the requested flexibility in two ways: *IRQ sharing* and *IRQ dynamic allocation*. In IRQ sharing, an interrupt handler executes several Interrupt Service Routines

[1] More information on this at Section 2.5.1.
[2] Note that an interrupt handler is part of a device driver.

(ISRs). Essentially, an ISR is a function related to one of the devices which share the IRQ line. Since the IRQ line is *shared*, it is not possible to know a priori which hardware device issued the interrupt. So, every ISR related to a device sharing the IRQ line is executed, in order to confirm if the corresponding device needs the CPU's attention. If that is the case, the ISR performs all the necessary work related to this particular interrupt.

In IRQ dynamic allocation on the other hand, each IRQ line is mapped to a device driver at the last possible moment. With IRQ dynamic allocation the same interrupt code or, more precisely, the same interrupt vector can be used by several hardware devices, even if they are not allowed to share the same IRQ line (e.g., in the ISA architecture). Apparently, these devices cannot be used at the same time.

Actions in I/O Interrupts

Until now, we know that an interrupt may require the execution of several actions associated with it. We also want the interrupt handlers to be extremely fast. These two conflicting needs form a big problem which has to be addressed with prudence in order to overcome it. Of course, not all the actions that need to be performed when an interrupt occurs have the same urgency. The solution for this problem is based on this exact idea, i.e. the classification of interrupt actions based on their urgency.

First of all, we have to make an important admission: the interrupt handler is not the place to execute all kinds of actions. Long, non-critical actions can and should be deferred, since, when an interrupt handler is running, all the interrupt requests in the corresponding IRQ line will be ignored. That is one of the reasons we want an interrupt handler to be fast. Also, the process which was running before the interrupt handler took over has to remain in runnable state (Section 2.2), so the interrupt handler must not execute any action that explicitly or implicitly blocks, e.g., reading from the HDD. If the reason for that is not apparent at first, consider the horrible events that will occur if there is not any task in runnable state: the system will freeze.

So, Linux divides the actions that have to be performed when an interrupt occurs into three main classes [Bove05]:

Critical: This class includes actions like acknowledging the interrupt to the Peripheral Interface Controller (PIC), reprogramming the PIC, or updating data structures accessed by the device and the processor. These actions can and will be executed quickly, inside the interrupt handler, and their prompt execution is an absolute requirement. They are executed with maskable interrupts disabled.

Non-critical: This class includes actions like updating data structures that are accessed by the CPU only, like reading the scan code after a keystroke. These actions must also finish quickly, so they are executed inside the interrupt handler, but with maskable interrupts enabled.

Deferrable: This class includes actions like copying the contents of a buffer into a process' address space (e.g., copying the keyboard line to the address space of the terminal). These actions can and will be deferred for a long period of time, without affecting any operations the kernel might be executing. This is allowed by the nature of these actions. Any process interested in the results of these actions, should just have to wait for them to finish. Deferrable actions are performed by means of deferrable functions, which are discussed in Section 2.4.

Let us now discuss the actual actions performed by the interrupt handler when an interrupt occurs. To begin with, there are some actions that are performed in all I/O interrupt handlers, irrespective of the device that caused the interrupt. The same four actions that are performed by all I/O interrupt handlers are described below:

1. Saving the IRQ code and the register's contents on the Kernel Mode Stack.
2. Acknowledging to the PIC servicing the IRQ line that the interrupt has been received, thus enabling it to issue more interrupts, if it desires so.
3. Executing all the ISRs of the devices that share the interrupt's IRQ line.
4. Terminate and return to the caller, by jumping to the address indicated by the function `ret_from_intr()`.

However, how is the kernel is able to distinguish among different types of interrupts? How is an interrupt represented internally?

In the kernel, an interrupt is represented using several descriptors. These descriptors represent both the state of an IRQ line, and the functions that should be executed when an interrupt occurs. So, when an interrupt occurs, both hardware circuits (e.g., the PIC), and software functions (e.g., `do_IRQ()`) are used to service the interrupt correctly. In general, each physical IRQ is assigned to an interrupt vector. Every interrupt vector has its own descriptor, called `irq_desc_t`, which has many fields, among which is the interrupt `depth`, the interrupt `lock` (which is used to serialize the accesses to the IRQ descriptor and the PIC), and the action list, which identifies the ISRs that need to be called when the IRQ occurs.

Now that we have a minimal understanding of the internal representation of interrupts, and the data structures associated with them, let's get back to the actions performed when an interrupt occurs. When interrupts are enabled, the receipt of an interrupt prompts a switch to *interrupt context*. More specifically, the kernel is in interrupt context when executing an interrupt handler or a bottom half. The function that is called to performed the top half, i.e. all the ISRs associated with an IRQ, is called `do_IRQ()`. Two big differences between the top half and the bottom half that should be clear by now, are the type of actions performed in each one, and the fact that interrupts are enabled during the execution of the bottom half. The second reason is also the main reason why bottom halves are deffered for a later, safer time. So, to sum up, a typical use case is presented below:

1. An interrupt occurs.
2. Switch to interrupt context is performed.
3. The kernel interrupt dispatch code retracts the IRQ number and the associated list of registered ISRs. The interrupt will be acknowledged by the proper ISR, and further interrupts issued from the same IRQ line will be ignored. The top half performs all the necessary actions like copying data to a buffer used by the device, schedules the bottom half handler, and returns. This series of actions is extremely fast.
4. After that, the bottom half will perform all the remaining actions that are necessary for servicing the interrupt, like waking up processes, performing I/O actions, etc.

This design enables servicing a new interrupt, while, for example, the bottom half is still working. Almost every interrupt handler is designed this way, and the interrupt handlers in the kernel's networking subsystem form a perfect example of this. When a network interface reports the arrival of a new packet, the interrupt handler just retrieves the data and forwards them to the protocol layer. The processing of the data will take place in a bottom half.

2.4 Bottom Halves and Deferrable Functions

In the last section we mentioned that during the execution of an interrupt there are plenty of actions that are non-critical, and thus can be deferred for a long time. We explained that the bottom half is scheduled to execute these actions, and we also mentioned that the ISRs in an interrupt handler are serialized and that, often, no interrupt should occur until the interrupt handler terminates.

This does not apply for a bottom half. As it was discussed earlier, deferrable actions can and will be executed with interrupts enabled, at a later, safer time. Removing these actions from the interrupt handler minimizes kernel's response time, and this is extremely useful for time-critical applications, which want their requests serviced as soon as possible (typically in a few milliseconds).

Linux kernels[3] use two ways to execute deferrable actions, in order to minimize its response latency:

Work Queues: these permit the execution of kernel function at later times, by some special threads called *worker threads*. These will not be discussed in this document (see [Bove05].)

Deferrable Functions: these include Softirqs and Tasklets, explained in depth later in this section. RCU relies heavily on Softirqs, but we will come back to that in Chapter 3. Deferrable functions are interruptible, and are often called *non-urgent functions*.

Softirqs and Tasklets are very close relatives since tasklets are essentially softirqs (or, more precisely, they are implemented on top of softirqs), however, they have some fundamental differences. To increase confusion, in kernel's source code, the term "softirq" often denotes both softirqs and tasklets. In the next paragraphs we will compare these two kinds of deferrable functions, and discuss the actions that can be performed on them. Also, remember that the term interrupt context (Section 2.3.3) means that the kernel is executing either a top or a bottom half.

The first main difference between softirqs and tasklets is that softirqs are statically allocated (i.e. they are defined at compile time), while tasklets can be dynamically allocated (e.g., when loading a kernel module). This is possible due to the fact that tasklets are implemented on top of softirqs. Another difference between these two types of deferrable functions is that softirqs are reentrant functions, while tasklets are not. Of course, this means that softirqs can be executed concurrently in different CPUs. However, it also means that softirqs have to explicitly protect their data structures using some synchronization mechanism (see Section 2.6), in this case, spinlocks. Since tasklets are not reentrant, they do not have to protect their data structures, but the execution of tasklets of the same type has to be serialized by the kernel. Although this means that the same tasklet type cannot be executed in two different cores, it does not forbid tasklets of different type to execute in different cores. Non-reentrant functions like tasklets greatly simplify the life of kernel driver developers.

2.4.1 Actions in Deferrable Functions

Let us see the four types of actions deferrable actions perform:

Initialization: Defines a new deferrable function. Initialization is performed on kernel initialization, or when loading a kernel module.

[3] Versions 2.6 and later.

Activation: Marks a deferrable function as *pending*. Pending deferrable functions will be executed the next time the kernel schedules to execute a batch of them. The activation can be performed at any moment, even within an interrupt handler (this is the case for RCU as well).

Masking: Selectively disables a deferrable function. This is done in order to prevent a deferrable function from executing, even if activated. Masking a deferrable function is useful at various points in the kernel.

Execution: Executes a batch of pending deferrable functions of the same type. The execution is performed at well-specified, pre-defined points in the kernel; see next section.

The activation and execution of a deferrable function are strictly correlated, in the sense that a deferrable function that is activated by a specific CPU must be executed by the very same CPU. This has not proved to be beneficial since there are contradicting arguments for both the execution of a softirq in the same CPU that it was activated, and the execution of the softirq in a different CPU. On the one hand, executing a softirq on the same CPU it was activated may make better use of the CPU's cache, but, on the other hand, since the execution is delayed for a long period of time, the data needed for the execution may not even reside in the CPU's cache anymore! Also, what happens if the CPU that activated the softirq is extremely busy, with frequent cache invalidations, while other CPU's are just sitting idle? Even though there is no apparent reason why executing a softirq in the same CPU it was activated is beneficial for the system, it is the way the kernel works, and many subsystems of the kernel -RCU included- rely on this fact.

2.4.2 Softirqs

In the previous section we discussed deferrable functions in general. In this section we will focus solely on softirqs. Softirqs is a very good starting point for establishing a solid understanding on deferrable functions and bottom half methods. Also, while tasklets are much easier to implement and use (since they do not need to be reentrant), they are built on top of softirqs. Lastly, a huge part of RCU relies on softirqs, and understanding how softirqs work will prove beneficial in understanding the underlying mechanisms of RCU. The code for softirqs in kernel resides in the file `kernel/softirq.c`.

Implementation and Data Structures of Softirqs

The enumeration type for all the existing softirq types is shown in Listing 2.7.

```
enum
{
        HI_SOFTIRQ=0,
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        BLOCK_SOFTIRQ,
        BLOCK_IOPOLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ,
        RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */

        NR_SOFTIRQS
};
```

Listing 2.7: Enumeration type for existing softirqs from `kernel/interrupt.h`.

A softirq's index determines its priority. Lower indexes indicate higher priority and vice versa, since the softirq functions will be executed starting from index 0.

The basic data structure used for the representation of softirqs is the `softirq_vec` array. This array's length is equal to 32 and the type of its elements is `softirq_action` (Listing 2.8). The priority of a softirq is essentially its index on this very array. Obviously, the length of the array dictates that only 32 (at most) different types of softirqs may exist, and, as it has already been mentioned, this number cannot change dynamically.

```c
struct softirq_action {
  void (*action)(struct softirq_action *); /* function to run */
  void *data;                              /* data to pass to function */
};
```

Listing 2.8: Structure representing a single softirq entry.

Listing 2.8 indicates that the data structure `softirq_action` has two fields: the action field, which is a pointer to the function that will be called when the softirq will be executed, and the data field, which is a pointer to a generic data structure that *may* be needed by the softirq's function.

Handling Softirqs

In Section 2.4.1 we stated that there are only four types of actions that can be performed on deferrable functions. Softirqs, being deferrable functions, have their counterparts, i.e. functions that perform the previously mentioned actions.

First of all, the initialization of a softirqs is performed via `open_softirq()` function. This function takes three arguments:

- the softirq's index,
- a pointer to the function this softirq will execute, and
- a second pointer to the data structure this function may need.

Note that `open_softirq()` only takes care of the softirq's initialization, i.e. the initialization of the respective entry in the `softirq_vec` array.

Then, there is softirq activation. This is taken care by the `raise_softirq()` function whose only argument is the softirq's index. After saving all the necessary flags and disabling interrupts, `raise_softirq()` marks the softirq as pending, by setting the bit corresponding to the softirq index, in the softirq mask of the CPU it is executed. After that, if it is not executed in interrupt context (this is determined by calling `in_interrupt()` function), `raise_softirq()` wakes up a special kernel thread called `ksoftirqd`, in the CPU it is executed. In the end, it restores the flags and returns.

Lastly, `do_softirq()` function is used in order to execute a softirq. A question that might have pop up by now is *when* are the softirqs actually executed, since we already mentioned that they are executed at well-specified, pre-defined points. Of course, if a CPU is executing an interrupt handler and there exist time consuming deferrable actions to be performed, a softirq will be raised. But this is not always the case. In general, how do we know that there are pending softirqs, and when does the check for pending softirqs has to be performed?

Well, the kernel handles these problems by performing periodic checks for pending softirqs. Naturally, this check must not add big overheads, and that is the reason it is performed at specific points in kernel code. The most important of these points are:

- When the kernel calls `local_bh_enable()` function to enable softirqs on the local CPU.
- When `do_IRQ()` function finishes handling an I/O interrupt and calls `irq_exit()`.
- If the system uses an I/O Advanced Programmable Interrupt Controller (APIC), when the `smp_apic_timer_interrupt()` function finishes handling a local timer interrupt. In this case as well, `irq_exit()` is called, which justifies our previous statement that timer interrupts behave like I/O interrupts.
- In SMP architectures, when a CPU finishes handling a function triggered by a `CALL_FUNCTION_VECTOR` IPI.
- When one of the `ksoftirqd` kernel threads is awakened.

Executing softirqs

If, in one of the above-mentioned points for pending softirqs, an activated softirq is detected, `do_softirq()` is called to handle it. This function, if called when in interrupt context, just returns. If that is not the case, it performs the following actions:

1. Calls `local_irq_save()` function to save the flags and disable interrupts on the local CPU. [4]
2. If it is necessary (we will not dig into details), it switches to the softIRQ stack.
3. Calls `__do_softirq()` function, which will perform all the actions related to this softirq.
4. Restores the original stack, if this has been switched in Step 2.
5. Calls `local_irq_restore()` to restore the flags, and returns.

The `__do_softirq()` function reads the local CPU's bitmask and executes the softirq functions which correspond to the set bits. However, due to the fact that other softirqs may be raised when softirqs are executed, something that could force `__do_softirq()` run for a long period of time, thus stalling user-space processes, `__do_softirq()` executes only a batch of irqs, while the remaining softirqs will be executed from a special kthread, the `ksoftirqd` kthread. This implies that every CPU has its own `ksoftirqd` kthread.

The implementation and usage of the softirqd kthreads represents the solution to a big trade-off: since softirqs are reentrant functions, and they may reactivate themselves, or even be activated by external events, what is the best way to deal with a large flow of softirqs?

One possible solution to deal with the potential high occurency frequency of the softirqs would be to execute only the softirqs which have been activated when `do_softirq()` started executing. However, this solution is prone to big delays in the execution of softirqs, introduced by the fact that softirqs will not be executed even if the CPU is idle. These delays are, of course, unacceptable to developers dealing with the kernel's networking subsystem.

Another possible solution would be to execute *all* the softirqs during the execution of `do_softirq()`, even those which are not activated at first. Apparently there is not any way that this can be made to work since, if, for example, a softirq re-activates itself[5], a system freeze may occur.

[4] Note that this does not contradict our previous statement that softirqs are executed with interrupts enabled. In Step 3, interrupts will be re-enabled when the softirq function will be executed, and then they will be disabled once again.

[5] Imagine what would happen if network packets arrived at the network card with a very high frequency and the respective softirq re-activated itself everytime a packet arrived.

Additionally, both of these solutions imply that softirqs will be served in timer interrupts. But what will happen if the CPU runs in tickless mode[6]? All of the above represent real problems kernel hackers needed to solve, and have done so. Using `ksoftirqd` kthreads enables the smooth operation of user-space processes (since only a batch of softirqs is executed at a time), and also enables the execution of softirqs in an idle CPU.

2.5 Time

Time is a crucial notion to the Linux kernel. Besides the fact that there are functions (or events) the actions of which are time-dependent, there are times when the kernel schedules functions to be executed at specific (future) times, and, of course, it must be able to know the current date and time. In the next subsections we will discuss the way the kernel counts and manages time.

2.5.1 HZ

The first question that comes to one's mind when thinking the kernel's interaction with time, is how the kernel is able to know that a time interval has passed. Or, to put it simply, how does the kernel count the time?

The answer to this question is that the kernel relies heavily on the system's hardware to "understand" the notion of time. The kernel uses the hardware's clock to understand how time goes by. The system's clock[7] is a simple integrated circuit (e.g., Intel 8253) and "ticks" (i.e. is activated) in a specific frequency, called the *tick frequency*. When the clock ticks, it emitts an interrupt which the kernel handles with a special interrupt handler. The handling of this interrupt resembles closely the handling of I/O interrupts, as we have already mentioned (Section 2.3.3).

The tick rate is programmed statically at the system boot, with the help of the C preprocessor and a simple define macro, and it is named `HZ`. That said, each timer interrupt will occur periodically, every `1/HZ` seconds. The value of `HZ` is architecture-dependent, e.g., x86 architectures define `HZ` to be equal to 100.

However, since the value of `HZ` is architecture-dependent, which value is optimal? Well, the answer to this question is non-trivial, and both lower and higher values have advantages and disadvantages.

Some advantages of high `HZ` values are:

- Higher interrupt frequency and therefore higher precision in the execution of time-dependent actions.
- More fine-grained process scheduling, since the scheduler is called in timer interrupts (at least in preemptive kernels), and the timeslot of a process decreases at every tick.
- Higher precision in measurements, e.g., resource usage, system uptime, etc.

while some advantages of low `HZ` values are:

- Less time is spent in interrupt handlers, since higher tick rate means more interrupts which the kernel is obliged to handle. This introduces a significant overhead in the kernel's workload.
- Less of the CPU's cache gets invalidated due to interrupt handling.

[6] Tickless operation is discussed in Section 2.5.3.
[7] We are not referring to the Real Time Clock (RTC).

- Smaller energy consumption, because the CPU can stay idle for longer periods of time (if, of course, has nothing else to do). Obviously, battery saving is a huge issue on laptops nowadays.

In general, choosing the right value for `HZ` is merely a design choice, differing from architecture to architecture.

2.5.2 Jiffies

Now that we have established a basic understanding on how the kernel counts the time, we will talk about `jiffies`. In the Linux kernel, the global variable `jiffies` is used to indicate the number of ticks in the system's uptime. On boot, this variable is initialized and it is incremented by one every time a timer interrupt occurs (i.e. at tick rate). Consequently, since there are `HZ` timer interrupts in a second, there are `HZ` jiffies in a second. To find the system's uptime (in seconds), we just have to divide `jiffies` with `HZ`. In `<linux/jiffies.h>`, we see that the variable `jiffies` is declared as: `extern unsigned long volatile jiffies;`.

To convert seconds to `jiffies` we can use the equation:

$$\texttt{jiffies} = \texttt{seconds} * \texttt{HZ}$$

whereas to convert `jiffies` to seconds we can use the equation:

$$\texttt{seconds} = \texttt{jiffies}/\texttt{HZ}$$

2.5.3 Tickless Operation

Now that we have described what the tick is, it is time to get rid of it. Well, not exactly, but timer interrupts do have some drawbacks. The biggest of them is the waste of processing power in interrupt handling, and the cache invalidation when this handling takes place.

In order to overcome these drawbacks, the Linux kernel supports *tickless operations*.

When the kernel is configured with the option `CONFIG_NO_HZ_IDLE` enabled (`CONFIG_NO_HZ` for older kernels), the scheduling-clock ticks on idle CPUs are omitted. After all, if a CPU is idle, there no use in sending it a timer interrupt. An idle CPU that does not receive scheduling-clock interrupts is said to be in *dynticks-idle* mode (or *nohz* mode for older kernels). This is the default options for most kernels.

When the kernel is configured with the option `CONFIG_NO_HZ_FULL` enabled, the scheduling-clock ticks are omitted on CPUs that are either idle or have only one runnable task. This is important for HPC applications and for real-time applications.

The obvious advantage of the tickless operation is that we can have the high frequency of interrupts, and the granularity they provide whenever we need it. When the completion of a specific process is considered very important, we can disable timer interrupts on a specified CPU (with the `CONFIG_NO_HZ_FULL` option) in order not to interrupt it. Additionally, some cores are permitted to be completely idle, something that promotes energy savings, and translates to longer battery life, e.g., in laptops.

Of course, there is a drawback, and in this case it is the increased complexity in the kernel's code. Writing code that operates correctly in tickless mode is a big challenge. Especially for RCU, the dynticks-idle mode added a lot of implications [Kernb].

The dynticks-idle mode was inaugurated in version 2.6 of the Linux kernel, and the full-dynticks mode was merged in kernel 3.10.

2.6 Synchronization Methods

Every developer of concurrent or parallel applications is familiar with race conditions. In order to make the developers' life easier, the kernel provides some mechanisms to enable them write fast, efficient code without races. In this section we will discuss some of these mechanisms, giving a brief description of the way they work. There are other locking mechanisms which are not covered in this thesis, e.g., rwlocks, seqlock, etc, for which the reader may refer to the kernel's Documentation (see [Kerna]).

2.6.1 Atomic Operations

The most fundamental synchronization method, of top of which other synchronization methods are built, is *atomic operations*. In parallel programming, there is often a need for an operation to be executed atomically, i.e. without interruption. In constrast with normal operations, it is not possible for two atomic operations to occur on the same variable concurrently. That said, increments and decrements cannot race. Of course, like many other synchronization methods in kernel, atomic operations are architecture-dependent, which means that while their interface is common, their implementation is different on every architecture that is supported in Linux.

There are three types of atomic operations:

- Atomic Integer Operations
- 64-bit Atomic Integer Operations
- Bitwise Atomic Operations

The first two types of atomic operation operate on special data structures (shown in Listing 2.9): `atomic_t`, and `atomic64_t`, respectively, while bitwise operations operate on generic memory addresses. However, all of these types' implementations are architecture-dependent. The `atomic_t` type implements 32-bit operations, while `atomic64_t` is its 64-bit counterpart. Their usage is exactly the same but `atomic64_t` should be used only in architecture-specific code (since not all CPUs support 64-bit operations) that requires operations to be performed on 64-bits. Following Linux's philosophy "portability over efficiency", `atomic_t` is the one that should be used.

```c
typedef struct {
        volatile int counter;
} atomic_t;

typedef struct {
        volatile long counter;
} atomic64_t;
```

Listing 2.9: Atomic data structures in Linux kernel.

The `atomic_t` types are used, even though they represent just a wrapped integer, because the use of the type system ensures that atomic operations are used with these types only. In addition, their usage prevents erroneous compiler optimizations (they are declared volatile), and unwanted typecasts. Lastly, while this may seem odd at first glance, these types hide any details in their implementation which is architecture dependent. That been said, it is now clear why we stated that their implementation hides (or, at least, used to) implementation details that are specific for each architecture. Different atomic functions are provided for a range of operations, and the full API is accessible in `<asm/atomic.h>`.

2.6.2 Spinlocks

Probably being the most common type of lock, *spinlocks* are used in a plethora of applications, and, of course, the kernel provides an API for using locks of this type. As expected, the implementation of spinlocks is architecture dependent. The API for using spinlocks is located in `<linux/spinlock.h>`, while the architecture-dependent code resides in `<asm/spinlock.h>`.

When multiple threads are contending for a lock, only one of them may acquire the lock, while the others are busy waiting, also called *spinning*. The same lock can be used in multiple locations, thus synchronizing accesses to a given data structure. Spinning is what defines the nature of a spinlock. The alternative behaviour would be to put the threads which failed to acquire the lock to sleep (see Section 2.6.3), and wake them up when the lock is available again. Spinlocks are meant to be a lightweight synchronization mechanisms used in cases where the lock should be held for a short period of time. Putting a thread to sleep and then waking it up requires two context switches, which, in case of a short critical section, are a big overhead. Especially in preemptive kernels, where the timeslice for each thread is consumed as time goes by, the duration the locks are held is equivalent to the scheduling latency of the system, thus making spinlocks the best choice for locks that should be held for short durations. However, unlike many spinlock implementations in user-space libraries, or other operating systems, in Linux, spinlocks are not recursive. Lastly, spinlocks can be used pretty much everywhere, including interrupt handlers (where, for example, semaphores or wait queues cannot be used) and bottom halves, making them a great choice for plenty of cases that require synchronized accesses to some data structure.

Reader-Writer Spinlocks

There are many cases where only writing demands mutual exclusion, while readers can execute concurrently. Imagine, for example, the manipulation of a list. Naturally, we want the updaters to run sequentially, with no concurrent updates, but we do not really care about how many readers are accessing the list, as long as there are is not any writer. So, for cases like this, where there are distinct reader and writer paths, we can use *reader-writer spinlocks*.

These locks provide two separate variants of a lock:

- The *reader lock*, which can be held by many readers simultaneously, and
- the *writer lock*, which can be held by at most one writer, with no readers concurrently accessing the data structure.

Like in most consumer/producer locking examples, one group has to be favoured in situations where a writer tries to update a data structure which is accessed concurrently by readers. In kernel, writers are not prioritized over readers, which means that if many readers are continuously accessing a data structure, a pending writer can starve.

2.6.3 Semaphores

We have already described the alternative behaviour to spinning, but we have not discussed the respective kernel mechanism that implements this behaviour. The older kernel mechanism that implements this kind of behaviour is called a *semaphore*. The idea for semaphores was conceived by Edsger Dijkstra in 1962 or 1963, and, fundamentally, they are sleeping locks. When a task tries to acquire a semaphore that is not currently available, the task is placed in a wait queue and then put to sleep. When the semaphore becomes available again, a task in the wait queue is awakened, but in the meantime, the processor can and will execute other code. The implementation code for semaphores is, of course, architecture dependent, and located in `<asm/semaphore.h>`.

An interesting feature of semaphores is that they allow the same lock to be held simultaneously by more than one threads. The number of the simultaneous holders is called *count*. If the count is equal to one, only one lock holder is permitted at a time, and the semaphore is called a *binary semaphore*. If the count is greater than one, the semaphore is called *counting semaphore*. As expected, binary semaphores can be used to enforce mutual exclusion (although this behaviour is not encouraged; see Section 2.6.4) and synchronization in general, whereas counting semaphores are used mostly for resource management.

Each semaphore supports two atomic operations: `down()` and `up()`. Operation `down()` is used to decrement the value of a semaphore, while operation `up()` is used to increment the value of a semaphore. If the new count value is zero or greater, the lock is acquired, but if the new value is negative, the task is placed on a wait queue.

Since is has become obvious that the big difference between spinlocks and semaphores lies in the way a thread behaves when it fails to acquire a lock, when does the usage of semaphores proves to be beneficial, in contrast to that of spinlocks? Well, first of all, if a thread needs to sleep, the use of semaphores is the only viable solution since spinlocks do not permit sleeping. On the contrary, if a lock is to be obtained in interrupt context spinlocks have to be used, because interrupt context is not schedulable. If neither of the above is the case, the decision on whether to use semaphores or spinlocks should be based on lock hold time: if short waiting periods are the case, spinlocks are the best choice, whereas semaphores are optimal for locks that are held for longer periods of time. Note, however, that semaphores do not add any scheduling latency (e.g., in non-preemptible kernels), while spinlocks might prevent a possible context switch.

Reader-Writer Semaphores

Semaphores also have their reader-writer variant. The cases where reader-writer semaphores should be used are exactly the same with their spinlock counterparts, i.e. there is clear distinction between the read and write path. One important detail, however, in the case of reader-writer semaphores, is the fact that all reader-writer semaphores are always binary semaphores.

2.6.4 Mutexes

The creation of mutexes originated from the need for a simple *mutual exclusion* sleeping lock. While binary semaphores were extremely popular, the fact that they do not enforce many usage rules made automatic debugging much harder. This is why mutexes were invented. Note that although the name *mutex* many times denotes a binary semaphore, in this document this is not the case, i.e. mutexes and binary semaphores are two completely different locking mechanisms, like in Linux kernel.

There are not many differences between binary semaphores and mutexes. However, despite the fact that their behaviour is quite similar, mutexes and binary semaphores do not have the same purpose. Mutexes are used for *exclusive access* to a resource, while binary semaphores should be used for synchronization (e.g., in event-driven programming). Binary semaphores should *not* be used for access restriction, since the acts of acquiring and releasing a semaphore are fundamentally decoupled. Typically, it does not make sense for the same task to release and acquire the same binary semaphore, since releasing a semaphore just notifies someone that is waiting to acquire it, that an event occured. Lastly, mutexes have a much simpler API and are faster than semaphores. The cases in which a mutex is preferred from a spin lock are pretty much identical with those in Section 2.6.3 regarding the choice between semaphores and spinlocks.

Runtime locking correctness validator

Unfortunately, all these wonderful locking mechanisms do not cleanse all the evil that comes with synchronization. In fact, locking has hazards of its own. Consider the repercussions of having two threads trying to acquire two distinct locks in different order. This will result in a deadlock, which, in turn, may result to a system freeze. The only way to avoid these type of deadlocks is to enforce some rules which define the order in which locks should be acquired. Of course, this is not trivial, since there are thousands of locks in Linux kernel, and the human error factor is not negligible.

Fortunately, there have been developed tools for automated code analysis in Linux kernel, such as the *lock dependency correctness validator*, or *lockdep* [Moln]. Although we will not go into details, this validator works by observing (or, better yet, tracking) real locking patterns in the kernel. But what are these locking patterns? Well, for example, every inode structure has a spinlock, and this spinlock has to be treated the same way for all the inode structures. This identifies a locking pattern. After identifying patterns, the validator code intercepts all locking operations and performs a number of tests like preventing a spinlock which is acquired by an interrupt handler to be held when interrupts are enabled, etc. Of course, all this checking causes overhead, but it is well worth it, since deadlock situations can be detected without having the system freeze or lock up.

2.6.5 Completions

The last synchronization mechanism that we will discuss is *completions*. Completions, also called *completion variables*, are used to implement synchronization between two tasks in the kernel, when one task needs to inform the other that an event for which the latter has been waiting on has occured. And while this does sound like a semaphore, there are some subtle but substantial differences.

To illustrate the most basic difference between semaphores and completions we will present an example. Consider two processes on a multiprocessor system: process A and process B. The following series of events introduces a race condition:

1. Process A allocates a semaphore, initializes the count to one, passes the semaphore's address to process B, and then invokes `down()` on it. Process A sleeps, waiting for an `up()` operations to happen. Then, as soon as the process awakens, it destroys the semaphore.

2. Process B, running on a different CPU, invokes `up()` on the same semaphore.

While this does not seem to introduce a race condition, the current implementations of `up()` and `down()` are not as innocent as they look. If these two operations are permitted to execute concurrently, process A can be woken up and destroy the semaphore, while process B is executing `up()`. This may result in process B attempting to access a data structure that no longer exists.

Such races does not exist in completions, and, other than that, completions just provide a simpler solution to a problem that can be solved with semaphores. The completion data structure is shown in Listing 2.10. The usage of a completion variable is simple: a task that needs to wait for an event calls `wait_for_completion()`, and when this event has occured the task calls the function `complete()`. These two functions are counterparts to the `down()` and `up()` semaphore functions.

When `complete()` is called, the `done` field is incremented, and the process waiting for this event in the wait queue is awakened. The above operations are done with the wait queue's spinlock held.

```
struct completion {
        unsigned int done;
        wait_queue_head_t wait;
};
```

Listing 2.10: Completion data structure.

When `wait_for_completion()` is called, the value of the `done` flag is checked. If it is greater than zero, the function returns (someone else has executed `complete()`). Otherwise, this task is added to the wait queue and is put to sleep, in `TASK_UNINTERRUPTIBLE` state, waiting for someone to awaken it. Again, these actions are performed with the wait queue's spinlock held.

The difference between semaphores and completions should now be clear: in completions, `complete()` and `wait_for_completion()` cannot execute concurrently, since the wait queue's spinlock is held across the body of these functions.

Chapter 3

Read-Copy-Update (RCU)

In this chapter we discuss the implementation and the idea behind Linux kernel's Read-Copy-Update (RCU) mechanism. We will discuss all the mechanisms RCU provides, as well as the way these mechanisms work. In addition, we will present the requirements every RCU implementation needs to satisfy, conclusions drawn from the usage of RCU, and, of course, the way a litmus test can be constructed, for the systematic checking of this mechanism.

3.1 Introduction to RCU

Read-Copy-Update (RCU) is a synchronization mechanism invented by Paul E. McKenney and John (Jack) D. Slingwine, and is a part of the Linux kernel as of 2002 [Wikic, McKe13, McKe04, Kernc, McKe]. The key feature of RCU is that it allows concurrent reads and updates, which renders it extremely scalable.

While this may seem counterintuitive or impossible at first, RCU achieves it in a very simple, yet extremely efficient way: by maintaining multiple data versions. RCU is carefully orchestrated in a way that not only ensures that reads are coherent and that no data will be deleted before pre-existing readers complete, but also uses efficient and scalable mechanisms which make read paths extremely fast. In fact, in non-preemptible kernels RCU imposes zero overhead to readers. Of course, updates can be relatively expensive, since updaters are weighted with the burden of maintaining older versions of data in order to accommodate pre-existing readers. It becomes apparent that, since there are multiple versions of the data, RCU is probably not applicable in situations where readers must not read outdated data (see Section 3.3.2). That said, although RCU is heavily used in many parts of the Linux kernel, the exact extent of its applicability is still a research topic [Trip09].

The basic idea behind RCU is to split updates in two phases [McKe07c]: the *removal phase* and the *reclamation phase*. During the removal phase, the updater *removes* references to data items either by destroying them (i.e. setting them to NULL), or simply by replacing them with references to newer versions of these items. This phase can run concurrently with reads due to the fact that modern microprocessors guarantee that a reader will see either the old, or the new reference to an object, and not a weird mashup of these two or a partially updated reference. During the reclamation phase, the updater frees the items removed in removal phase, or, in other words, these items are *reclaimed*. Of course, since RCU allows concurrent reads and updates, the reclamation phase must begin *after* the removal phase and, more specifically, when it is certain that there are no readers accessing or holding references to the data being reclaimed.

So, the typical update procedure using RCU looks like the following [McKe98]:

0. Ensure that all readers accessing RCU-protected data structures carry out their references from within an RCU read-side critical section.

1. Pointers to a data structure are removed, so subsequent readers cannot gain a reference to it (removal phase).

2. Wait until all pre-existing readers complete their RCU read-side critical section, so there no one holding a reference to the item being removed.

3. At this point, there cannot be any readers still holding references to the data structure, so it now may safely be reclaimed, e.g., freed (reclamation phase).

Note that steps 1 and 3 in the aforementioned procedure are not neccessarily performed by the same thread.

Step 2 in this procedure is the basic idea behind RCU and the deferred destruction of data. The ability to wait until all pre-existing readers have completed their RCU read-side critical sections is what enables readers to use lightweight synchronization, or, in some cases, no synchronization at all. Waiting for pre-existing readers can be achieved either by blocking, or by registering a callback that will be invoked after all pre-readers have completed their RCU read-side critical sections.

A typical update procedure using RCU is depicted in Figure 3.1.

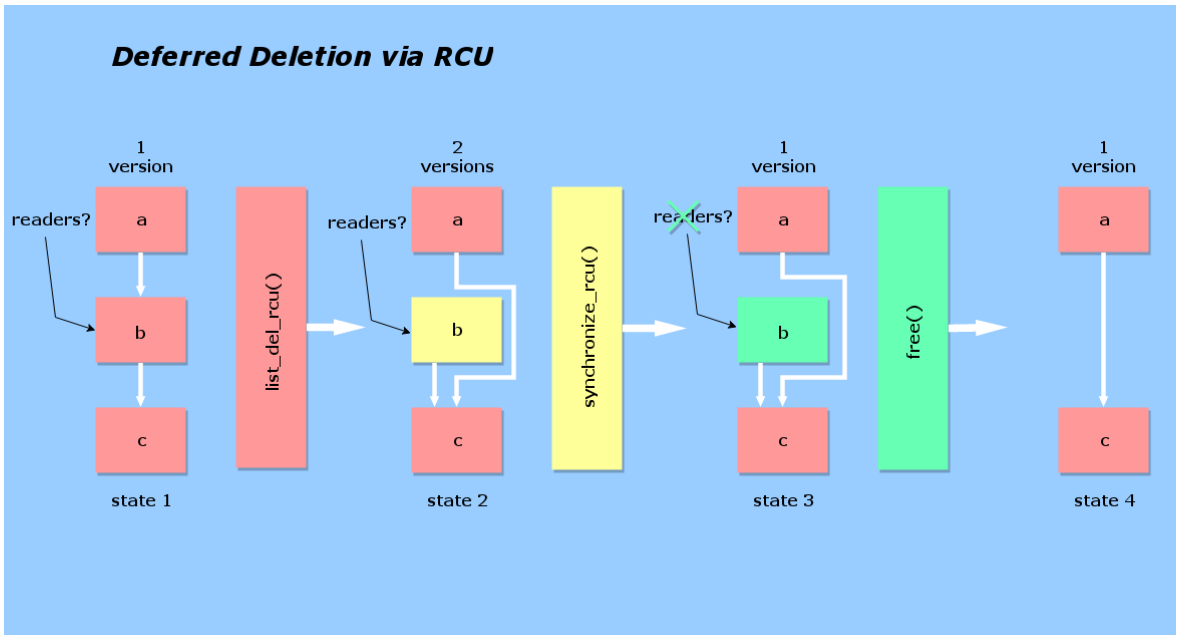


Figure 3.1: Deferred Deletion via RCU (adapted from [McKe13]).

3.2 RCU Fundamentals

RCU consists of three basic mechanisms [McKe07c]: one for inserting data, one for waiting for pre-existing readers to complete, and one for maintaining multiple versions of data. These mechanisms are described in the following subsections, along with examples which should be helpful in understanding the way these mechanisms work.

3.2.1 Publish–Subscribe Mechanism

As mentioned, the key feature of RCU is the fact that it allows concurrent reads and updates, or, otherwise stated, that updaters can modify some data while readers are scanning these exact data. RCU achieves that by providing an insertion mechanism, which greatly resembles a Publish–Subscribe mechanism. The best way to understand this mechanism is through a simple example.

Consider a global pointer `gp`, which is initialized to `NULL`. Suppose that this pointer is modified by an updater to point to a newly allocated and freshly initialized data structure. One code fragment that does the aforementioned is presented in Listing 3.1.

```
1   struct foo {
2     int a;
3     int b;
4   };
5   struct foo *gp = NULL;
6
7   /* . . . */
8
9   p = kmalloc(sizeof(*p), GFP_KERNEL);
10  p->a = 42;
11  p->b = 42;
12  gp = p;
```

Listing 3.1: Publish reordering effects.

However, there is absolutely *no guarantee* that the three last statements will be executed in order. In Chapter 2, it was made clear that it is within the rights of both the compiler and the CPU to reorder these statements in the name of efficiency. Having said that, the events that induce such a reordering are not self-evident. So, it would be worthwile to elucidate the way such a reording might take place.

First of all, suppose that the compiler holds the value of `gp` in a register. If the compiler needs to spill that value in order to generate the code for lines 10-12, it might as well generate the code for line 12 *before* it generates the code for lines 10-11 (assuming of course that the memory locations for `p` and `gp` do not overlap). Remember that the compiler is free to perform any optimizations it wants, if they do not alter the behaviour of a single-threaded program. This is why a compiler directive (compiler barrier, or the usage of **`volatile`**, if appropriate) is needed to maintain the desired ordering.

Moreover, even if the compiler generates the code in order, the CPU might trip up the code in different ways. Even if the CPU executes the instructions in order, the value of `gp` can be flushed from the store buffer long before the values of the assignments of lines 10-11 will be flushed (e.g., if the cache line referenced by `p` is not owned by the CPU executing these statements, while the cache line referenced by `gp` is). However, there are also plenty of cases where the CPU routinely executes code out of order, e.g., in superscalar CPUs. In these cases, the reasons why these statements can be reordered become self-evident.

Now, suppose that a reader locklessly picks up the value of the `gp` pointer and, if it is non-`NULL`, locklessly accesses the fields `a` and `b`. In that case, assuming that the assignments of lines 10-12 are reordered, a reader may see uninitialized values for both the fields of the `gp` pointer.

In order to avoid the adverse effects of reading uninitialized values, memory barriers should be used. RCU offers the `rcu_assign_pointer()` primitive, which is similar to an assignment statement but also provides additional ordering guarantees. More specifically, `rcu_assign_pointer()` has similar semantics to C11's `memory_order_release` operation, but also prevents "nasty" compiler optimizations. With `rcu_assign_pointer()`, the last three lines of Figure 3.1 would be as follows:

```
p->a = 42;
p->b = 42;
rcu_assign_pointer(gp, p);
```

Listing 3.2: Publish mechanism.

In the above code snippet, the `rcu_assign_pointer()` macro *publishes* the new structure, forcing both the compiler and the CPU to perform the assignment to the `gp` pointer only *after* the fields of the data structure have been initialized.

Of course, similar problems affect the readers as well. Consider, for example, the following code snippet:

```
p = gp;
if (p != NULL) {
  /* do something with p->a, p->b */;
}
```

Listing 3.3: Subscribe reordering effects.

Although at first glance the above code fragment does not seem like it can cause problems, a more careful look reveals that this is not the case. Value speculation optimizations performed by the compiler, as well as dependent loads reordering performed by some CPUs (like the notorious DEC Alpha), can cause the values of the fields `a` and `b` to be fetched before the value of `p`, or can cause the values `p->a` and `p->b` to be read from different structures. Another way the compiler may trip up this code is presented below:

```
if (gp) {
  /* do something with gp->a, gp->b */
}
```

Listing 3.4: Subscribe reordering effects – optimized code.

Note that in the above code snippet, the compiler refetches from `gp` and does not use `p` at all! This may be the case when the compiler is short of registers. And while refetching from `gp` may seem as a reasonable choice, if this code snippet ran concurrently with an updater that replaced the current version of the data structure with a new one, the reader could have read a different value for the fields `a` and `b`, if the fetches come from different data structures. So, in order to avoid these problems (and many others, e.g., load tearing [Howe]), RCU provides the `rcu_dereference()` primitive:

```
rcu_read_lock();
p = rcu_dereference(gp);
if (p != NULL) {
  /* do something with p->a, p->b */
}
rcu_read_unlock();
```

Listing 3.5: Subscribe mechanism.

The `rcu_dereference()` primitive encapsulates all the compiler and memory barriers required for its purpose. It can be considered as a *subscription* to a value of a specified pointer, and it guarantees that subsequent dereference operations will see any initialization that took place before the `rcu_assign_pointer()` (publish) operation. The `rcu_dereference()` primitive has semantics similar to C11's `memory_order_consume` load, and uses both volatile casts and memory barriers (for DEC Alpha), in order for it to provide the aforementioned guarantee.

Using `rcu_assign_pointer()` and `rcu_dereference()` is required for programs that access RCU-protected pointers. Moreover, if a pointer is RCU-protected (annotated with `__rcu`), all dereferences of that pointer have to be performed inside RCU read-side critical sections using the `rcu_dereference()` primitive, and all assignments to RCU-protected pointers have to use the `rcu_assign_pointer()` primitive. A tool called `sparse` is used in the Linux kernel [Spar] in order to ensure (among other things) the correct manipulation of RCU-protected pointers, and to avoid bugs related to RCU usage. RCU read-side critical sections are delimited by the `rcu_read_lock()` and `rcu_read_unlock()` macros. These macros do not block or spin, and in non-preemptible kernel they are effectively no-ops.

Lastly, RCU provides an API for easier list manipulation (e.g., iterate over a list, insert an element to a list, delete an element, replace an element, etc.) for both double-linked lists and circular lists, data structures used heavily in the Linux kernel. We will not present the full API here, although interested readers may find useful information at the following sources: [McKe08b, McKe10b, McKe14].

3.2.2 Waiting for Pre-Existing Readers to Complete

In the previous section, it was mentioned that for a data structure to be safely freed, RCU has to wait for all the pre-existing readers to complete their RCU read-side critical sections. But the way RCU does that is not in any case obvious to an unfamiliar reader. In this section we will discuss the way RCU knows that all pre-existing readers have completed their read-side critical sections, and we will present a toy implementation for this mechanism.

Essentially, RCU is *a way of waiting for things to finish*; it can be considered a bulk reference counting mechanism. The things RCU waits on are *RCU read-side critical sections*, and RCU can wait for thousands of RCU read-side critical sections to complete, using a very simple, yet extremely scalable way.

RCU read-side critical sections start with `rcu_read_lock()` and end with `rcu_read_unlock()`. These macros can be nested, and RCU treats a nested set as one big RCU read-side critical section. However, these macros are not permitted to block or sleep[1]. In effect, RCU can wait for *any* piece of code to complete, as long as this piece of code is located within an RCU read-side critical section.

As we have already discussed (Section 3.1), the typical update procedure using RCU resembles the following:

1. Update a data structure (e.g., replace an element in a double-linked list via the `list_replace_rcu()` function).

2. Wait for all pre-existing RCU read-side critical sections to complete. This can be done in various ways. One way is to call `synchronize_rcu()` (we will discuss this primitive later on), which will block until all pre-existing readers have completed their critical sections. However, note that once all pre-existing RCU read-side critical sections have completed, there is no way for a reader to gain a reference to the removed element.

3. Reclaim, e.g., free the replaced element.

In the example above, there are two facts worthy of attention. First, RCU read-side critical sections starting *after* `synchronize_rcu()` was called, can complete after `synchronize_rcu()` has returned. Second, when waiting for `synchronize_rcu()` to return, there might be readers seeing both the old and the new element in that position of the list. This fact will be discussed further in the next section.

The typical update procedure discussed above can be better understood with the example presented in Listing 3.6. In this example, the last three lines represent the typical update procedure. The updater searches the list for a specified element and, if the element is found, it *copies* the element (while there might be concurrent *readers*) and then executes an *update*.

[1] A special flavour of RCU, called Sleep RCU (SRCU), permits sleeping within RCU read-side critical sections [McKe06].

```
struct foo {
  struct list_head list;
  int a;
  int b;
};
LIST_HEAD(head);

/* ... */

p = list_search(head, key);
if (p == NULL) {
  /* ... */
}
q = kmalloc(sizeof(*p), GFP_KERNEL);
*q = *p;
q->a = 42;
q->b = 42;
list_replace_rcu(&p->list, &q->list); /* Modifies an element in the list */
synchronize_rcu();                    /* Waits for pre-existing readers */
kfree(p);                             /* Frees the replaced element */
```

Listing 3.6: Typical RCU update example.

Overall, the only issue not addressed thus far is the way tha RCU is able to know that all pre-existing read-side critical sections have completed. And the fact that the `rcu_read_lock()` and `rcu_read_unlock()` primitives do not generate any code in non-preemptible kernels is definitely puzzling. After all, how is synchronization without changing machine state possible?

Well, the key here is that RCU read-side critical sections are not permitted to block or sleep[2]. So, when a CPU executes e.g., a context switch, any prior RCU read-side critical sections will have completed. Consequently, as soon as each CPU has executed at least one context switch, all pre-existing readers are guaranteed to have completed their RCU read-side critical sections, and `synchronize_rcu()` can safely return. An optical illustration of the above is presented in Figure 3.2. Though there are much more to this than the aforementioned, these are sufficient for a basic understanding of this RCU mechanism.

In order to formalize some of the aspects presented above, it would be useful to provide some definitions.

Definition 3.1. Any statement that is not within an RCU read-side critical section is said to be in a *quiescent state*.

Such statements are not permitted to hold references to RCU-protected data structures (in Linux kernel, this can be checked with `sparse`). Note that different RCU flavours have different sets of quiescent states. For example, for Tree RCU-sched (the main flavour of RCU used in the kernel, see Section 5.1), quiescent states include:

- Context switch
- Idle mode (idle loop or dynticks-idle), and
- User-mode execution

while for Tree RCU-bh (a flavour that can withstand DDoS attacks) quiescent states are any code outside of softirqs with interrupts enabled. Of course, the wait-for-readers mechanism

[2] This applies for both the Tree RCU and the (now obsolete) Classic RCU flavours. There are other RCU flavours that permit sleeping or preemption within RCU read-side critical sections ([McKe06, McKe07a]).

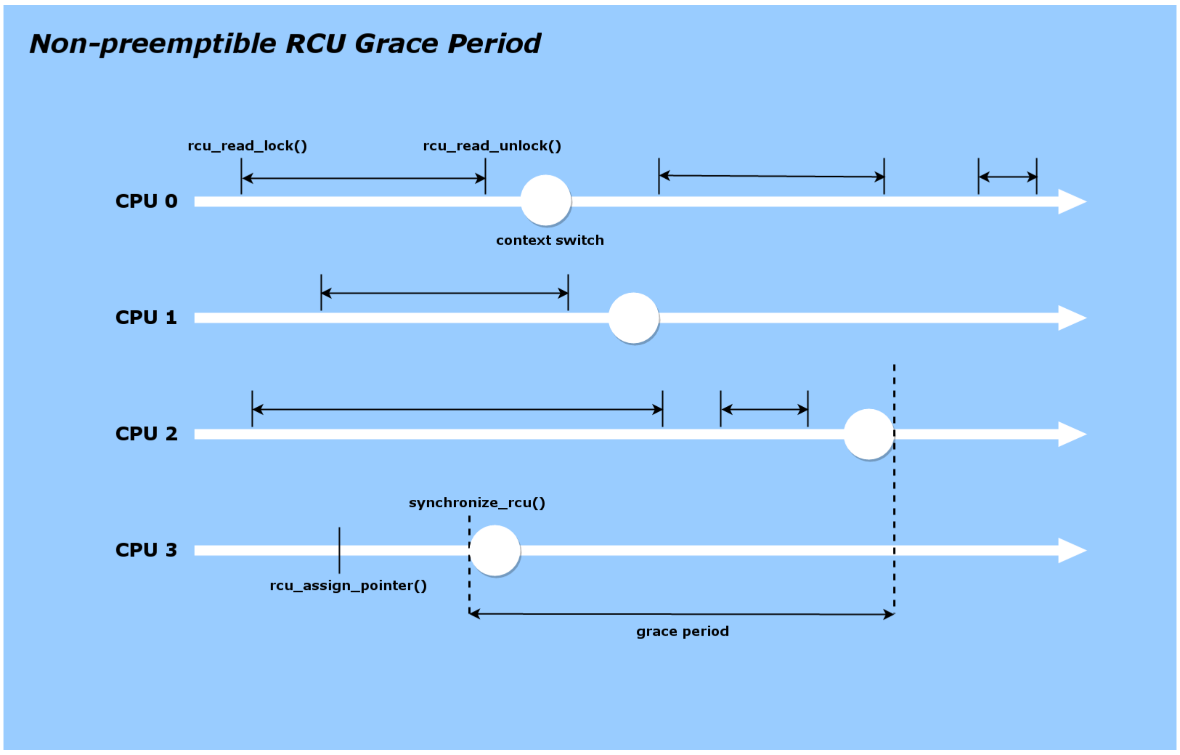


Figure 3.2: Waiting for pre-existing RCU readers to complete (adapted from [McKe13]).

is not required to wait for CPUs that are in quiescent state; it is required to wait only for pre-existing readers.

Definition 3.2. *Any* time period during which each CPU resides at least once in a quiescent state is called a *grace period*.

Consequently, if an RCU read-side critical section had started before the beginning of a specified grace period, it has to complete before the end of that grace period. This means that the wait-for-readers mechanism has to wait for *at least* one grace period to elapse. Once a grace period has elapsed, there can no longer be any readers holding references to the old version of a newly updated data structure (since each CPU has passed through a quiescent state) and the reclamation phase can safely begin.

Given these points, a toy implementation of `synchronize_rcu()` would be as follows: `for_each_online_cpu(cpu) { run_on(cpu);}`. In this code, `run_on()` just runs the current thread to the CPU specified (of course, this forces a context switch on that CPU, but in a place where it is permitted). As can been seen, this loop forces a context switch on each CPU, something that guarantees that all pre-existing RCU read-side critical sections have completed. Note that this implementation would not work for kernels that use real-time preemption. That said, the implementation for `synchronize_rcu()` in Tree RCU (which is the main RCU flavour in the kernel) does not look like this trivial implementation at all. The actual implementation in the Linux kernel is much more complex, as it is required to handle many scenarios (including interrupts, NMIs, CPU hotplugs, full dynticks-idle mode, etc.), and it differs depending on the RCU flavour, the kernel version, etc. Be that as it may, a conceptual implementation of `synchronize_rcu()` is very useful, since the basic idea of this mechanism remains the same, and it is the one presented in the previous paragraphs. Lastly, note that RCU also offers an asynchronous primitive for the wait-for-readers mechanisms, called `call_rcu()`. This primitive registers a callback that will be invoked after all pre-existing readers have completed their RCU read-side critical sections, i.e. after at least one grace period has elapsed.

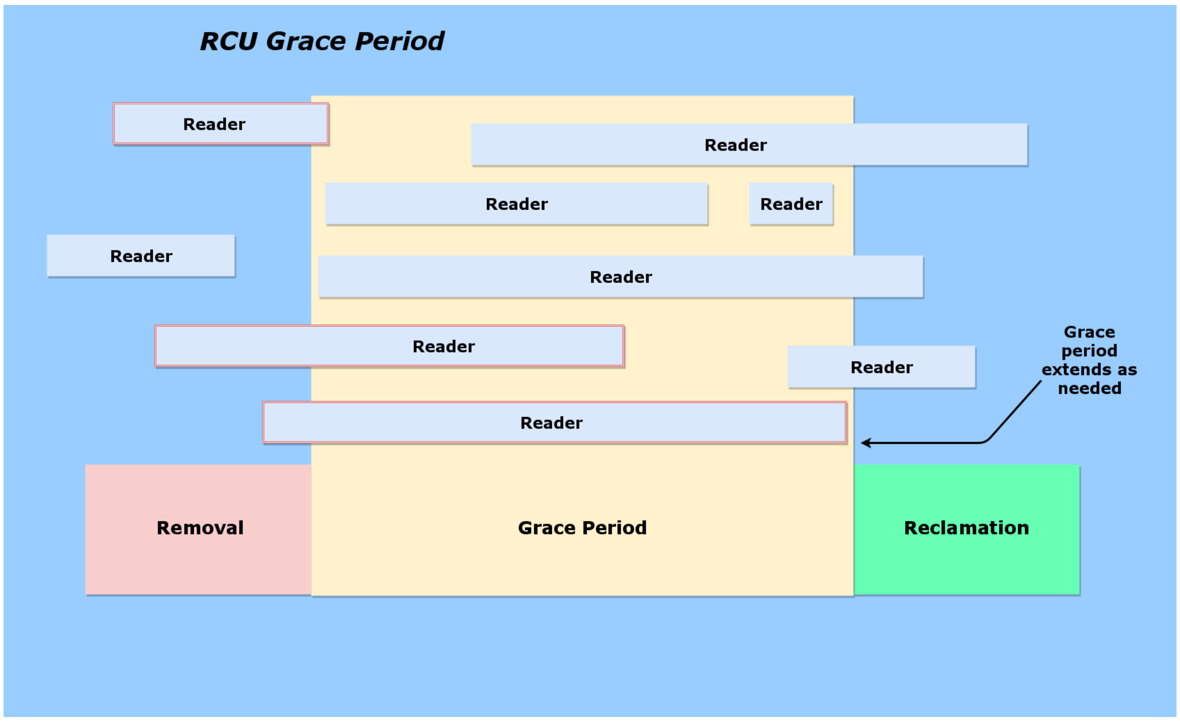


Figure 3.3: The reclamation phase starts after a grace period has elapsed (adapted from [McKe07c].

3.2.3 Maintaining Multiple Versions of Recently Updated Objects

The third and last fundamental RCU mechanism we will discuss, addresses the issue of what RCU readers see when they are traversing a concurrently updated list. Although the basic notions may already be apparent from Section 3.1, it is not always obvious how an element referenced by a specified reader remains intact while an updater is concurrently modifying it. We will present and discuss examples in the hope that in the end of the current section, a reader will have an adequate understanding of how RCU works, at least conceptually.

First of all, let us consider the typical update procedure example given in Section 3.2.2. Note that this example can be modified to demonstrate the deletion of a list element, with just minor changes and the usage of `list_del_rcu()` primitive. While this example may seem simple at first glance, there are some points that are easy to miss.

In general, it should be obvious that there might be two versions of an element visible to concurrent readers. For example, a reader may fetch a pointer to an element and then take an interrupt. In this case, if an updater copies and updates this element, the reader may see the old version of the list for a decent amount of time after the removal. So, in a case like this, we have two versions of the list.

What is not so obvious though, is *how* the `list_replace_rcu()` primitive replaces an element, without disrupting concurrent readers. In the case of a double-linked list the `list_replace_rcu()` primitive is presented below (kernel v3.19):

```c
1  static inline void list_replace_rcu(struct list_head *old, struct list_head *new)
2  {
3          new->next = old->next;
4          new->prev = old->prev;
5          rcu_assign_pointer(list_next_rcu(new->prev), new);
6          new->next->prev = new;
7          old->prev = LIST_POISON2;
8  }
```

Listing 3.7: Definition for `list_replace_rcu()`.

where the `LIST_POISON2` macro is a non-`NULL` pointer that will result in page fault under normal circumstances. Lines 4-5 just set the `->prev` and `->next` pointers of the newly inserted elements to the proper values. Line 6 sets the `->next` pointer of the predecessor of the newly inserted element to point to the newly inserted element. Line 7 sets the `->prev` pointer of the successor of the newly inserted element to point to the newly inserted element, and line 8 poisons the `->prev` pointer of the replaced element. But, what would happen if there were readers traversing the list both forwards and backwards? Would not the last line cause a `SEGFAULT`?

Well, the answer to this question is that readers that are traversing an RCU-protected list have to use primitives provided by RCU for that (e.g., `list_for_each_entry_rcu()`, etc.). These primitives traverse a list in a specific order (forwards only) and avoid this kind of issues. Essentially, the `->prev` pointer poisoning helps in detecting readers that traverse the list in the opposite direction. And since RCU permits more than one versions of a list to be active at a given time and updaters run concurrently with readers, there is little to no use in trying to keep the list consistent for bi-directional readers (e.g., with atomic two-pointer updates). However, should there ever be a need for RCU-protected traversal of linked lists both backwards and forwards, the RCU implementation could easily change in order to accommodate it.

3.3 Conclusions Drawn from the Usage of RCU

Now that a basic understanding of RCU and its internal mechanisms has been established, we will discuss advantages and disadvantages that emerge from the usage of RCU. And while many of these may already be apparent and do not introduce anything new, the way one thinks about the RCU and the mechanisms it provides is crucial for what we will do in the next chapters. After all, RCU is much more than a simple synchronization mechanism, and this will become apparent in the next paragraphs.

3.3.1 Some Remarks

As noted in Section 3.2.2, RCU is a way of waiting for things to finish [McKe08c]. Therefore, RCU can be considered a reference-counting mechanism. Consider the code snippet shown in Listing 3.8. In this code fragment, `synchronize_rcu()` waits for all previously acquired references to `p` to be released, since all these references reside within RCU read-side critical sections. Note that there cannot be any new references to `p`, due to the assignment to `head`. When no thread holds a reference to `p` anymore, the updater is free to delete it. Of course, this reference counting scheme is somewhat limited since RCU read-side critical sections do not block or sleep. Moreover, a reference cannot be passed from one thread to another.

```
spin_lock(&lock);
p = head;
rcu_assign_pointer(head, NULL);
spin_unlock(&lock);
synchronize_rcu();
kfree(p);
```

Listing 3.8: RCU is a reference-counting mechanism.

However, in contrast to traditional reference counting mechanisms, RCU is not associated with a specific data structure or data structure group, and thus does not need to maintain a global reference counter for the aforementioned data structure/group. As a result, RCU can be thought as a bulk reference-counting mechanism with extremely low overhead, due to RCU's extremely efficient and scalable read-side primitives. So, whereas a traditional reference counting mechanism would suffer from severe performance degradation when the read-side usage for a specific data structure group increases, RCU's read-side primitives guarantee that the read-side overhead will be trifling; and this is the main reason as to why one should use RCU instead of a traditional reference-counting scheme, if, of course, the circumstances allow it (e.g., there is no need for a reference to be passed from a task to another).

But, since RCU can be considered a reference counting mechanism, and these mechanisms can be used to implement garbage collectors, can RCU be considered a garbage collector too? Well, RCU does resemble a reference-counting based garbage collector, however there are some subtle, yet important differences: the programmer has to specify the section within which references can be held (i.e. the RCU read-side critical sections) and also has to manually specify when a data structure can be collected. This means that, while there are many similarities, a garbage collector based on RCU would *not* be an automatic one. Other than that, RCU can be considered a garbage collector, and it can provide guarantees that either an object is not referenced anymore, or that an object remains in existence (for the duration of an RCU read-side critical section).

Lastly, we will discuss the various interactions between the readers and the updaters, as well as the ordering constraints imposed by RCU. The first point one should keep in mind is that readers interact with updaters only via the grace-period API. This means that readers cannot prevent an updater from modifying a data structure; but they can prevent a grace period from ending. Besides, it has already been pointed out that updaters run concurrently with readers. Another important point that needs to be considered is that the RCU read-side critical section primitives provide absolutely no guarantee with regard to the instruction ordering. After all, in some cases they are no-ops. What this means though, is that any ordering constraints within an RCU read-side critical section should be imposed by the programmer; otherwise, the code is susceptible to both compiler optimizations and possible adverse effects of the employed memory model.

3.3.2 Comparison with Other Synchronization Mechanisms

One should argue that there many ways of waiting for things to finish. After all, we have already mentioned that RCU resembles a reference counting mechanism; and there other ways too: reader-writer locks, hashed locks, events, hazard pointers, etc. However, it is not always obvious when RCU can be used, and which are its advantages and its limitations, especially over conventional synchronization mechanisms. Thus, this section attempts to clarify these matters.

So, can RCU be used in cases where other synchronization mechanisms were used?

Well, most conventional synchronization mechanisms offer some kind of mutual exclusion between readers and updaters, or allow concurrent reads, but not concurrent reads and writes (see Section 2.6). Of course, there exist synchronization mechanisms that allow readers and writers to operate concurrently on an object (e.g., seqlocks), but these mechanisms force the updaters to repeat their action everytime this occurs. However, in cases where concurrent reads are permitted (for example, in reader-writer locks), it should be possible to use RCU instead. The only questions remaining are *why bother* and *what implications* does the usage of RCU have.

Ultimately, what makes RCU stand out are its advantages in terms of scalability and efficiency, i.e. its *performance*. RCU does not perform the updates in-place like many conventional locking mechanisms, and takes advantage of the fact that modern microprocessors guarantee that writes to single aligned pointers are atomic, which permits the modification of data structures without disrupting any concurrent readers. This way, concurrent readers can avoid the indubitably expensive cost of memory barriers, atomic operations and cache misses, and continue to access an older version of a data structure (if, of course, this is acceptable algorithm-wise). RCU scales extremely well in cases where the read-side critical sections are small (thus inducing small overhead), since readers do not contend for a global lock.

In addition, RCU read-side critical sections are immune to deadlocks since their primitives do not block, spin, or do a backwards jump instruction. Moreover, this very fact means that the read-side critical section primitives provide real-time latency and have deterministic execution time, which renders RCU much more scalable than other synchronization mechanisms.

Lastly, the fact that readers and updaters run concurrently with RCU means that the readers can see the new values in updated data structures soon. In contrast, if a different synchronization mechanism that permitted only concurrent reads were used, an updater could not have updated the value until all pre-existing readers would have ended their critical sections, and readers trying to read the value after an updater tried to acquire the lock would fail, since they would spin until the updater has ended its critical section. However, these subsequent readers in the case of a reader-writer locking scheme are guaranteed to see the updated value, whereas in the case of RCU, readers may or may not see the updated value. Note that even pre-existing RCU readers may see the updated value, if their RCU read-side critical section is long, and they acquire the reference after the value has been updated. That said, only the readers that begin their RCU read-side critical sections after the updater has finished the update section are guaranteed to see the update value.

At this point it should be apparent that the usage of RCU comes at a price: there will be internal inconsistencies and the readers may read stale data [Arca03, McKe04]. In contrast, other synchronization mechanisms like reader-writer locks guarantee internal consistency (though in these cases, there may be increased inconsistency with the outside world). Moreover, in RCU, readers block updaters, in the sense that updaters have to wait for pre-existing readers to complete. This means that it is possible for a low-priority process to block a high-priority process for a long time, given that RCU grace periods last for milliseconds. In order to avoid this, updaters have to use RCU priority boosting or asynchronous primitives like `call_rcu()` that do not block.

As can be seen, RCU fits best in situations that include read-mostly workloads. But the question remains: when is RCU best used? This question will be answered in the next paragraphs.

Areas that RCU Applies to

RCU works best in read-mostly workloads, where stale and inconsistent data are acceptable. A classic example in the Linux kernel is the networking routing table. Since updates can take a considerable amount of time to reach a given system (seconds or even minutes, since they have to propagate across the Internet), the system will have been sending packets the wrong way for quite some time. So, a few additional milliseconds are not normally a problem; and we already noted that RCU readers might see the updated values sooner than the readers in a reader-writer locking scheme would.

Now if the workload is read-mostly or read-write, but consistent data is required, RCU is applicable as well, though it has to be used in conjunction with some other synchronization primitives like per-data-structure locks, per-thread locks, seqlocks, atomic operations, etc. For example, if a data structure that maps from a key to data structures is used and these data structures need to be synchronized, RCU can be used to protect the mapping data structure only, and a per-structure lock can be used to protect the data structures being mapped to. A per-structure deletion flag has to be used as well to indicate if the desired operation can be performed. The operation will be performed under the protection of the structure's lock, and the updaters will have to wait for a grace period before they free the data structure.

Unfortunately, in cases where the workload is update-mostly, and consistent data is required, the RCU is probably not applicable (also see [Arca03]); in these situations, RCU is used usually to provide existence guarantees only (i.e. as long as someone references a data structure, it will not be deleted).

3.4 RCU Specifications

In this section we will present some requirements that every RCU implementation must fulfill. This section is not a formal specification of RCU, and RCU requirements have been presented in the past as well [McKe15a, McKe15b, McKe15c, Kernd]; however, it will serve as a guide, regarding the type of tests that need to be run in order to verify some properties of RCU.

3.4.1 Grace-Period Guarantee

The fact that in RCU updaters wait for all pre-existing readers to complete their read-side critical sections constitutes the only interaction between the readers and the updaters. The Grace-Period guarantee is what allows updaters to wait for all pre-existing RCU read-side critical sections to complete. As a reminder, RCU read-side critical sections start with `rcu_read_lock()` and end with `rcu_read_unlock()`.

What this guarantee means, is that the RCU implementation must ensure that any read-side critical sections in progress at the start of a given grace period will have completely finished (including memory operations, etc.) before that grace period completes. This very fact allows RCU validation to be focused; every implementation has to adhere to the following rule:

> If any statement in a given RCU read-side critical section precedes a grace period, then all statements -including memory operations- in that RCU read-side critical section must complete before that grace period ends.

Memory operations are included in order to prevent the compiler or the CPU from undoing work done by RCU.

In order to see what this guarantee really implies, consider the following code fragment where all variables are initially zero:

```c
int x;
int y;

int r_x;
int r_y

void thread_reader(void)
{
  rcu_read_lock();
  r_x = READ_ONCE(x);
  r_y = READ_ONCE(y);
  rcu_read_unlock();
}

void thread_update(void)
{
  WRITE_ONCE(x, 1);
  synchronize_rcu();
  WRITE_ONCE(y, 1);
}
```

Listing 3.9: RCU's Grace-Period guarantee example.

Since `synchronize_rcu()` has to wait for all pre-existing readers to complete their RCU read-side critical section, the outcome:

```
r_x == 0 && r_y == 1
```

is impossible. This is what the Grace-Period guarantee is all about. It is the most important guarantee that RCU provides; in effect, it constitutes the core of RCU.

Another litmus test for the Grace-Period guarantee is presented in Listing 3.10.

```c
int r_x;
int r_y;

int x = 0;
int y = 0;

void thread0(void)
{
        WRITE_ONCE(x, 1);
        synchronize_rcu();
        r_y = READ_ONCE(y);
}

void thread1(void)
{
        rcu_read_lock();
        WRITE_ONCE(y, 1);
        r_x = READ_ONCE(x);
        rcu_read_unlock();
}
```

Listing 3.10: RCU's Grace-Period guarantee example 2.

In this example, if after both threads have completed the outcome

```
r_x == 0 && r_y == 0
```

were possible, this would indicate a failure of RCU.

Finally, note that in both examples above, `call_rcu()` could have been used in place of `synchronize_rcu()` if we did not want the updater to block. Of course, RCU read-side critical sections must refrain from calling `synchronize_rcu()` (or wait on the completion of that function), since this may cause deadlocks.

3.4.2 Publish-Subscribe Guarantee

The Publish-Subscribe guarantee is used in order to coordinate read-side accesses to data structures. In Section 3.2.1, we discussed that the Publish-Subscribe mechanism is used in order for data to be inserted into data structures (e.g., lists), without disrupting concurrent readers.

As a matter of fact, we have already explained what this guarantee is about in Section 3.2.1. We have seen both the Publish and the Subscribe mechanism (Listings 3.2 and 3.5 respectively), along with examples of what might happen if these two mechanisms are not used. In other words, the Publish-Subscribe guarantee is provided by the Publish-Subscribe mechanism, relies on `rcu_assign_pointer()` and `rcu_dereference()` primitives, and ensures that RCU readers will have a consistent view of newly added data elements. The example in Listing 3.11 shows a simple publish-subscribe scenario where the reader is guaranteed not to see uninitialized values. In this example, the reader is guaranteed to see the initialized values for the field values of `p`, i.e.:

$$\texttt{p->a == 42 \&\& p->b == 42} \tag{3.1}$$

The Publish-Subscribe guarantee dictates that a different outcome from the one above should be impossible.

3.4.3 Unconditional Read-to-Write Upgrade

In general, RCU does not impose any kind of restrictions on readers, meaning that it should always be possible for a reader to perform updates within an RCU read-side critical section. Of course, readers cannot invoke `synchronize_rcu()` within an RCU read-side critical section since this may cause problems like deadlocks (`call_rcu()` is one way to avoid deadlocks, as we have already mentioned), but other than that, a reader is free to carry out updates within its critical section. The result of such an action if other concurrent updaters are present depends on the synchronization mechanism used by the updaters (e.g., whether writers use a global or multiple spinlocks, atomic operations, task designated updates, etc.). That said, it is the programmer's concern to keep the writers from interfering with each other in a disruptive manner.

3.4.4 Unconditional Execution of the RCU Primitives

The last guarantee provided by RCU is that the RCU primitives are unconditional, meaning that they perform the necessary work and exit. I.e., that they cannot fail.

The implementations for some of the most common RCU primitives can be found at `<linux/rcupdate.h>`, while some update-side primitives reside in the respective RCU implementation file.

```c
bool add_gp(int x, int y)
{
        struct foo *p;

        p = kcalloc(1, sizeof(*p), GFP_KERNEL);
        if (!p)
                return -ENOMEM;
        spin_lock(&gp_lock);
        if (rcu_access_pointer(gp)) {
                spin_unlock(&gp_lock);
                return false;
        }
        p->a = x;
        p->b = x;
        rcu_assign_pointer(gp, p);
        spin_unlock(&gp_lock);
        return true;
}

bool use_gp(void)
{
        struct foo *p;

        rcu_read_lock();
        p = rcu_dereference(gp);
        if (p) {
                BUG_ON(p->a != 42 || p->b != 42);
                /* do something with p->a, p->b */
                rcu_read_unlock();
                return true;
        }
        rcu_read_unlock();
        return false;
}

void *thread_publisher(void *arg)
{
        add_gp(42, 42);
        return NULL;
}

void *thread_subscriber(void *arg)
{
        use_gp();
        return NULL;
}
```

Listing 3.11: RCU's Publish-Subscribe guarantee example.

Chapter 4

Verification of Tiny RCU

In this chapter we will mechanically validate the Grace-Period guarantee for Tiny RCU, a flavour of RCU designed to run on uniprocessor systems. Although this is not the first mechanical proof of this guarantee (Paul McKenney presented the first proof for Tiny RCU in 2015, see [McKe15d]), it will be the first step towards the verification of some portions of Tree RCU for non-preemptible kernels, which will be presented in Chapter 5. The code used for all the tests used in both Chapters 4 and 5 is located at `https://github.com/michalis-/rcu`, tag: `diploma_thesis`.

The layout of this chapter is as follows: in Section 4.1 we discuss some implementation information regarding Tiny RCU, in Section 4.2 we present some definitions used to emulate a Linux-kernel uniprocessor (UP) environment, and in Section 4.3 we present a litmus test for the correct operation of Tiny RCU.

4.1 General Information

Tiny RCU is an RCU implementation for uniprocessor systems, designed to have significantly smaller memory consumption, making it ideal for embedded systems [McKe09b]. The key feature that distinguishes Tiny RCU from other RCU implementation is the following:

> Any time the sole CPU of the system passes through a quiescent state, a grace period has elapsed.

This feature greatly simplifies the design of Tiny RCU's implementation since it enables a smaller memory footprint and simpler RCU-related data structures. This fact also implies that the pending callbacks of a CPU could be invoked on each context switch; however, this is not the case neither for Tiny, nor for Tree RCU. The core of RCU in both of these flavours is executed in softirq context (also see Section 5.1), since it is desirable, in general, for RCU callbacks to be able to call functions related to scheduling (e.g. `wake_up()`), without problems like deadlocks.

Although we have already presented a high-level explanation of all the key concepts of RCU, it is important to discuss some implementation details crucial for RCU's operation. For the next paragraphs, it is important to keep in mind that Tiny RCU is only offered for non-preemptible kernels.

First of all, how does RCU know that a processor has passed through a quiescent state? Well, Tiny RCU relies on context switches, scheduling-clock interrupts and idle mode, just like Tree RCU, albeit in a slightly different way. Note that this is not a rule but rather an implementation choice; quiescent states could be manually noted by the programmer (see QSBR flavour of user-space RCU [Desn09, Desn12]), but this is not the case here. An efficient and scalable implementation is of great significance.

Every time a CPU takes a scheduling-clock interrupt, `rcu_check_callbacks()` is called (`rcu_check_callbacks` must be called from hardirq context). This function checks if the CPU is in a quiescent state (i.e. if the CPU is in user mode execution or interrupted from idle), and, if so, calls `rcu_sched_qs()` to inform RCU[1]. Now, `rcu_sched_qs()` marks the pending callbacks of the CPU as ready to invoke (by updating the callback list) and raises a softirq. The ready callbacks of a CPU will be executed in softirq context from `rcu_process_callbacks()`. The callback list is split into segments, because `rcu_process_callbacks()` needs to execute only the callbacks that are ready to be invoked and not callbacks that have been -possibly-registered after the quiescent state but before the softirq execution.

Then again, this is only half the story. When the CPU enters idle mode, `rcu_sched_qs()` is called since idle mode is both a quiescent state and a grace period for Tiny RCU, and this call will have the effects described in the previous paragraph. The above effectively summarize the way callbacks are processed in the case of Tiny RCU.

However, `synchronize_rcu()` works in a slightly different way, as can be seen from the following code:

```
void synchronize_sched(void)
{
        rcu_lockdep_assert(!lock_is_held(&rcu_bh_lock_map) &&
                           !lock_is_held(&rcu_lock_map) &&
                           !lock_is_held(&rcu_sched_lock_map),
                           "Illegal synchronize_sched() in RCU read-side critical section");
        cond_resched();
}
```

Listing 4.1: `synchronize_sched()` implementation for Tiny RCU.

This function does not rely on RCU callbacks and `call_rcu()`, as opposed to Tree RCU; its implementation is rather simple. In effect, a call to function `synchronize_rcu()` is a quiescent state since it is illegal to call this function from an RCU read-side critical section. The function `cond_resched()` is used for explicit rescheduling in places that are safe, and may invoke the `__schedule()` function (see Section 2.2), which marks a quiescent state by calling `rcu_note_context_switch()`, which in turn calls `rcu_sched_qs()`. The fact that a call to `synchronize_rcu()` is a quiescent state, constitutes the main reason why `cond_resched()` is called.

4.2 Kernel Environment Modeling

It is self-evident that RCU relies on many of the kernel's subsystems for its correct operation. Tiny RCU includes many kernel files and uses many primitives from the kernel. Moreover, we have to find a way to emulate the environment of a non-preemptible UP system and the restrictions it imposes. All these pose interesting problems that will be addressed in the next paragraphs.

4.2.1 CPU, Interrupts & Scheduling

CPU

Since Tiny RCU operates on a non-preemptible UP system, only one thread runs on the CPU at a time. Consequently, we need some kind of mutual exclusion between threads contending for the CPU. This mutual exclusion can be provided by a mutex, and since the tests will be run under Nidhugg, which supports the pthreads library, a `pthread_mutex` was used.

[1] `rcu_bh_qs()` can also be called from `rcu_check_callbacks()`, for the operation of RCU-bh. However, we will ignore all the RCU-bh-related functions for simplicity.

But how does one thread acquire the CPU? Well, the answer is presented in Listing 4.2.

```
void fake_acquire_cpu(void)
{
        if (pthread_mutex_lock(&cpu_lock))
                exit(-1);
        rcu_idle_exit();
}
```

Listing 4.2: Fake CPU acquisition for Tiny RCU.

The code in Listing 4.2 implies that the CPU starts out idle. Indeed, the sole CPU in the system is idle at first, and exits the idle mode when a thread acquires the CPU's lock. This may not be the case in a real system, but for our modeling this approach is sufficient. In a similar manner, when a thread finishes its execution, it calls `fake_release_cpu()`, as shown in Listing 4.3.

```
void fake_release_cpu(void)
{
        rcu_idle_enter();
        if (pthread_mutex_unlock(&cpu_lock))
                exit(-1);
        if (need_softirq) {
                need_softirq = 0;
        }
}
```

Listing 4.3: Fake CPU release for Tiny RCU.

Note that just before a thread releases the CPU, the CPU enters idle mode. The `need_softirq` variable indicates whether a softirq has been raised. We did not model softirqs for Tiny RCU, although it is not hard to do so; see Section 5.2. Hence, asynchronous primitives like `call_rcu()` cannot be used in our tests. However, this should not pose a problem. Our tests rely on `synchronize_rcu()` primitive, because Tiny RCU has a different implementation for this primitive than Tree RCU. Should one want to test `call_rcu()` in place of `synchronize_rcu()`, it will not be hard to do so, e.g. by following a similar approach with the one described in Section 5.2 for Tree RCU.

Interrupts

As for the interrupt handling, there are two locks: the `irq_lock` indicating that a thread has its interrupts masked, possibly due to being in an interrupt handler, and the `nmi_lock` indicating that a thread is in an NMI handler[2]. These locks are just pthread mutexes. Note that a thread cannot acquire the `cpu_lock` or the `irq_lock` while holding the `nmi_lock`, and a thread can acquire the `irq_lock` only after acquiring the `cpu_lock`. A counter is also used in order to accommodate nested interrupts and functions like `local_irq_save()` and `local_irq_restore()`.

Scheduling

Lastly, as shown in the previous section, context switches are quiescent states for Tiny RCU. In addition, we noted that `cond_resched()` is used in the `synchronize_rcu()` primitive.

[2] Although this way Non Maskable Interrupts can be modeled, these are not involved in our tests.

Obviously, this function needs to be modeled as well, keeping in mind that `synchronize_rcu()` can block. Listing 4.4 presents the way `cond_resched()` is modeled.

```
void cond_resched(void)
{
        fake_release_cpu();
        fake_acquire_cpu();
}
```

Listing 4.4: `cond_resched()` function for Tiny RCU.

In the kernel, this function invokes the scheduler only if there is a need for rescheduling. If the scheduler is called, it is noted that the CPU has passed through a quiescent state. In order to model this scenario, our version of `cond_resched()` just drops the CPU's lock and then (possibly) re-acquires it. A quiescent state is marked as well, courtesy of `rcu_idle_enter()`, called from `fake_release_cpu()`.

4.2.2 Kernel Definitions

Tiny RCU includes many files from different subsystems of the kernel, and we want to keep the RCU source code intact in order to test it. One way to overcome this problem is to add the test directory to the compiler's include path and provide empty files in place of the files included, while keeping the necessary definitions in files of our own, which will be included in our testing files. Some of the necessary definitions were copied directly from the Linux kernel v3.19 (e.g. `IS_ENABLED()`, `bool`, `barrier()`), while others were stubbed (e.g. preemption control, boot tests, softirq activation, etc.) or faked (e.g. `cond_resched()`). Note that the `WARN_ON_ONCE()` macro and its relatives (e.g. `BUG_ON()`) have been transformed into `assert()` statements, something that proved beneficial both later in this chapter and in the next.

4.3 Validating Tiny RCU's Grace-Period Guarantee

To mechanically prove Tiny RCU's Grace-Period Guarantee, we can use a test similar to the one presented in Listing 3.9. The relevant portion of a suitable test is presented in Listing 4.5.

As we have already mentioned, the result:

$$\texttt{r\_x == 0 \&\& r\_y == 1} \tag{4.1}$$

should be impossible. Since we do not want this condition to be violated, we can simply add a similar assertion statement at the end of the test.

One can run the test (provided in the github repository) under sequential consistency with the following command:

```
nidhuggc -I . -std=gnu99 -- --sc --disable-mutex-init-requirement fake.c
```

where `fake.c` is the file containing the litmus test.

It only takes Nidhugg 0.08 seconds to claim that no assertion is violated in this test. But is this claim valid? In any case, there might as well be a bug in Nidhugg, our test, or in the emulated environment for the Linux kernel.

This is where the `FORCE_FAILURE` conditional group in Listing 4.5 comes to play. What this group does is inject a bug in the reader's critical section which can cause the Assertion 4.1 to fail. So, if there is at least one interleaving that can cause the assertion to fail, Nidhugg should be able to find one such interleaving.

```c
int x, y;
int r_x, r_y;

void *thread_update(void *arg)
{
        fake_acquire_cpu();

        x = 1;
        synchronize_rcu();
        y = 1;

        fake_release_cpu();
}

void *thread_reader(void *arg)
{
        fake_acquire_cpu();

        rcu_read_lock();
        r_x = x;
#ifdef FORCE_FAILURE
        rcu_read_unlock();
        cond_resched();
        rcu_read_lock();
#endif
        r_y = y;
        rcu_read_unlock();

        fake_release_cpu();
}
```

Listing 4.5: Litmus test for validation of Tiny RCU's Grace-Period guarantee.

Running Nidhugg again, but with `-DFORCE_FAILURE` passed in the `C` flags, we see that Nidhugg does find one such interleaving:

1. The reader acquires the CPU and executes `r_x = x`.
2. The reader executes `cond_resched()` and releases the CPU but does not re-acquire it yet.
3. The updater acquires the CPU and executes `x = 1`.
4. The updater executes `cond_resched()`, releases the CPU and immediately re-acquires it.
5. The updater executes `y = 1` and terminates.
6. The reader re-acquires the CPU from step 2, executes `r_y = y`, releases the CPU and terminates.

Obviously, the aforementioned sequence of events does violate the assertion 4.1.

Additionally, Nidhugg needs only 0.08 seconds to find a counterexample for this test as well, in contrast to CBMC [Clar04] (v4.9) which needs 8.55 and 10.10 seconds to claim success and failure, respectively.

Finally, due to the fact that RCU interacts with the dynticks-idle subsystem via the dyntick-idle counters, and because the relevant portion of the kernel's source code contains `WARN_ON_ONCE()` statements (which have been transformed into `assert()` statements), this test serves as a mechanical validation of some properties of these counters as well. In general, `WARN_ON_ONCE()` statements are present in Tree RCU as well, and their replacement with `assert()` statement was extremely helpful in determining whether our modeling of the kernel was sound. Of course, Tiny RCU has a rather trivial implementation (only 389LOC in v3.19). The verification of Tree RCU is significantly more challenging, but more on this in the next chapter.

Chapter 5

Verification of Tree RCU

In this chapter, we will mechanically validate the Publish-Subscribe guarantee (Section 5.4) and the Grace-Period guarantee of Tree RCU (Section 5.5). In addition, we will reproduce an existing bug (Section 5.3.1), and show that a previously reported bug does not qualify as a bug (Section 5.3.2). But first, we will provide a high-level explanation of the implementation of Tree RCU (Section 5.1), and explain the way the kernel's environment was modeled in order for our tests to run (Section 5.2).

The code for both this chapter and Chapter 4 is located at `https://www.github.com/michalis-/rcu`, tag: `diploma_thesis`.

5.1 General Information

The Linux kernel offers many different RCU implementations, each one serving a different purpose. The first RCU implementation (available since 2002 and Linux kernel 2.5.43) was Classic RCU. The problem with Classic RCU was that, despite the fact that its read-side primitives were extremely efficient and scalable, the primitives used to determine when all pre-existing readers have finished their read-side critical section had limited scalability. More specifically, Classic RCU suffered from increasing lock contention due to the presence of one global lock that had to be acquired from each CPU that wished to report a quiescent state to RCU.

However, this was not the only problem. From Linux kernel 2.6.21, dynticks-idle mode was merged into the kernel (`CONFIG_NO_HZ_IDLE=y` or `CONFIG_NO_HZ=y` for older kernels). This option omits scheduling-clock ticks on idle CPUs since reducing OS jitter is important for HPC applications and for real-time applications [Kernb]. Unfortunately, Classic RCU had to wake up every CPU at least once per grace period, thus preventing idle CPUs from entering deep C-states[1], and increasing power consumption [Inte]. In the case where updates are frequently performed in a few CPUs with the rest CPUs being in idle-mode (e.g., in systems sized for peak loads), a significant amount of energy was unnecessarily wasted.

Tree RCU offers a solution to both these problems since it reduces lock contention and avoids awakening dyntick-idle CPUs [McKe08a]. Tree RCU scales to thousands of CPUs easily, while Classic RCU could scale only to several hundred.

In the next subsections, we will present a high-level explanation of Tree RCU along with some implementation details (the implementation of Tree RCU greatly varies between different kernel versions), a brief overview of Tree RCU's data structures, and some use cases that will help one understand the way RCU's fundamental mechanisms are actually implemented.

[1] C-states apply to Intel processors only. AMD processors use different power saving technologies like Cool'n'Quiet [AMDC].

5.1.1 High-Level Explanation

As we have already mentioned, in Classic RCU each CPU had to clear its bit in a field of a global data structure (namely `rcu_ctrlblk`) after it had passed through a quiescent state. Since CPUs operated concurrently on this data structure, a spinlock was used to protect the mask (`->cpumask`), something that, of course, may suffer from extreme contention.

Tree RCU addresses this issue by creating a heap-like node hierarchy, thus greatly reducing lock contention. The key here is that CPUs will not try to acquire the same node's lock when they are trying to report a quiescent state to RCU; in contrast, CPUs are split into groups and each group will contend for a different node's lock. Each CPU has to clear its bit in the corresponding node's mask once per grace period. The last CPU to check in (i.e. to report a quiescent state to RCU) for each group, will try to acquire the lock of the node's parent, until the root node's mask has been cleared. This is when a grace period can end (we will examine the exact way this happens in Section 5.1.3). A simple node hierarchy for a 6-CPU system is presented in Figure 5.1.

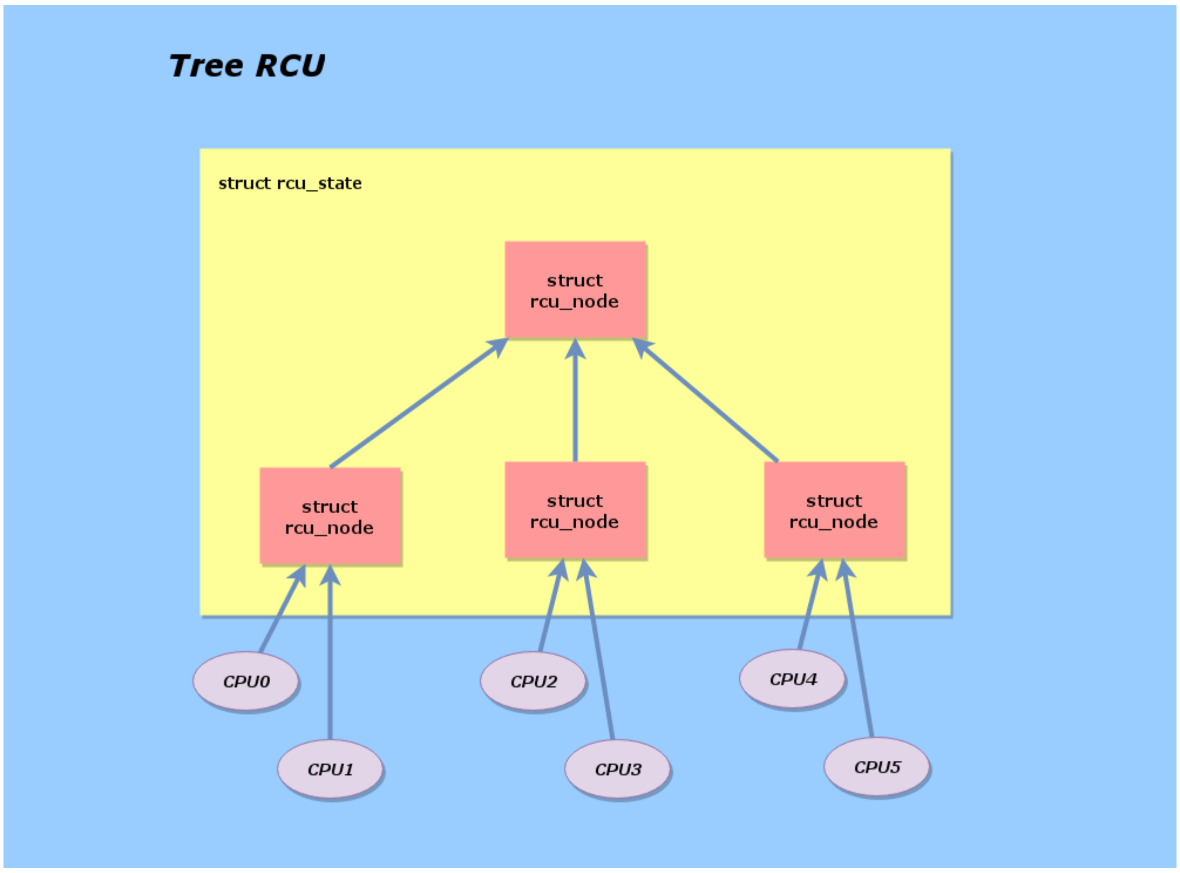
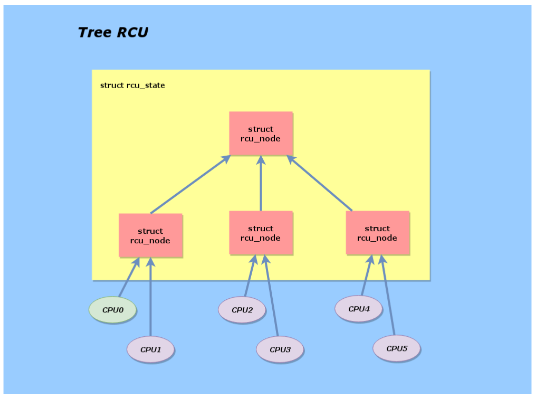


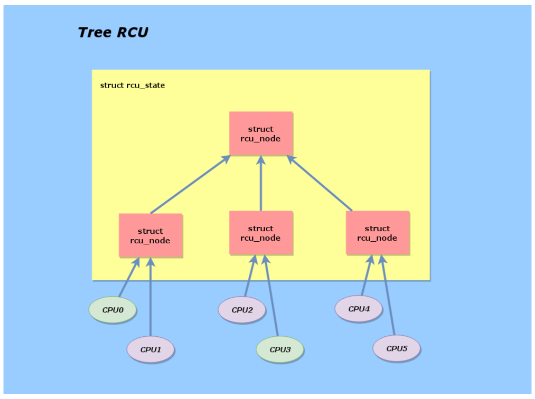
Figure 5.1: Tree RCU node hierarchy (adapted from [McKe08a]).

As can be seen from Figure 5.1, CPU0 and CPU1 will acquire the lower-left node's lock, CPU2 and CPU3 will acquire the lower-middle node's lock, and CPU4 and CPU5 will acquire the lower-right node's lock. The last CPU reporting a quiescent state for each of the lower nodes, will try to acquire the root node's lock, and this procedure happens once per grace period.

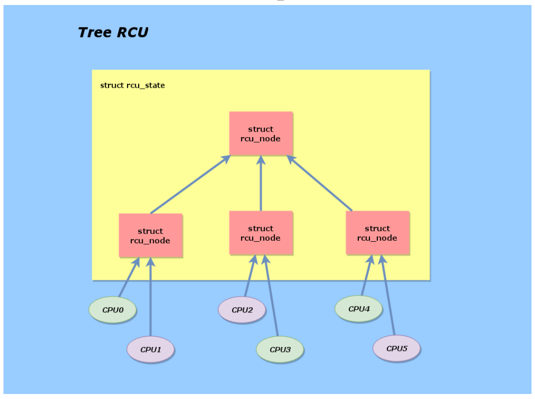
In order for this procedure to become more clear, consider the example in Figure 5.2. As can be seen, CPUs 0, 3 and 4 pass through a quiescent state and report it to RCU (i.e. clear their bit in the corresponding node's mask). Then, at the same time, CPUs 1, 2 and 5 report a quiescent state to RCU and, since they are the last in their group, they will attempt to acquire the root node's lock. Of course, this means that they will be serialized and will acquire the lock in a specific order. When all CPUs have passed through a quiescent state at least once, the grace period can end.



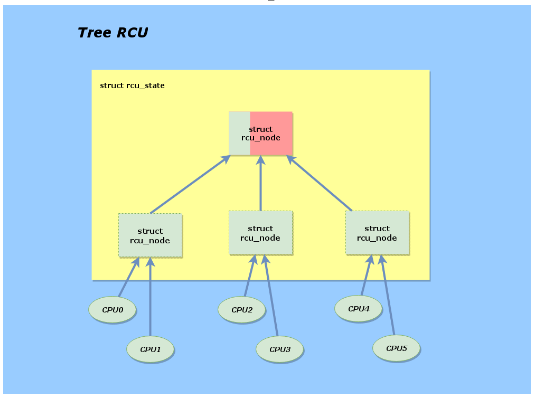
(a) CPU0 reports a QS.



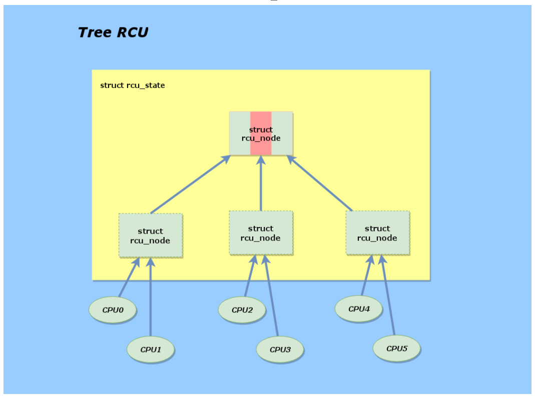
(b) CPU3 reports a QS.



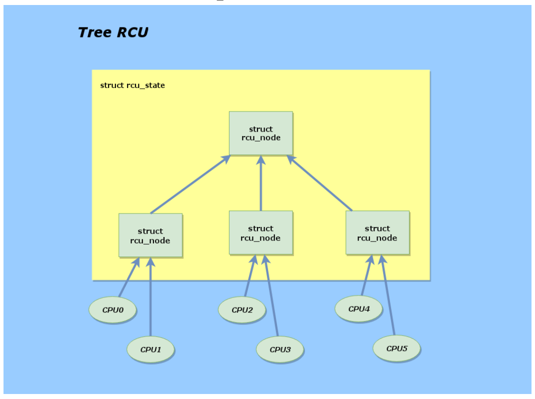
(c) CPU4 reports a QS.



(d) CPU1 acquires the root's lock.



(e) CPU5 acquires the root's lock.



(f) CPU2 acquires the root's lock.

Figure 5.2: A simple hierarchy example (adapted from [McKe08a]).

In comparison with Classic RCU, the hierarchical nature of Tree RCU results in much lower lock contention: at most three CPUs contend for a lock during each grace period in this example (only two CPUs contend for the leaf-nodes), whereas in Classic RCU as many as six CPUs would contend for the `rcu_ctrlblk`'s lock during each grace period. And while this may not seem like a big deal, in systems with a large number of CPUs, the reduction in lock contention is even bigger. For example, for a 16,384-CPU system, only 16 CPUs will contend for the lower node structures, and at most 32 CPUs will contend for the inner node structures, assuming a multi-level hierarchy. This implies a dramatic reduction from 16,384 CPUs contending for the `rcu_ctrlblk` structure's lock in Classic RCU.

Finally, the node hierarchy created by Tree RCU is fully tunable, and is controlled by the `Kconfig` options. There are two options, namely:

`CONFIG_RCU_FANOUT_LEAF`: Controls the leaf-level fanout of Tree RCU, i.e. the maximum number of CPUs contending for a leaf-node's lock. Default value is 16.

`CONFIG_RCU_FANOUT`: Controls the fanout of Tree RCU, i.e. the maximum number of CPUs contending for an inner-node's lock. Default value is 32 for 32-bit systems and 64 for 64-bit systems.

More information can be found at the `init/Kconfig` file. Note that a big hierarchy needs more time for grace-period initialization and cleaning up, and there are more structures that need scanning when a grace period needs to be forced (see Section 5.1.3). Other than that, the user is free to configure the hierarchy in any way desirable, and RCU can currently -in theory- accomodate 16,777,216 CPUs on 64-bit systems (of course, some modifications would be needed should a system that large existed).

5.1.2 Data Structures Involved

Notwithstanding the aforementioned high-level explanation of Tree RCU, a validation of correctness for the fundamental RCU mechanisms requires a deep understanding of the implementation of Tree RCU. In this section, we will attempt to briefly present some important aspects of the implementation of Tree RCU, crucial for the development of test cases aiming at the validation of some fundamental RCU mechanisms.

The most essential RCU mechanism is the wait-for-readers mechanism, as we have thoroughly mentioned throughout this document. And though we have provided a high-level explanation for this mechanism, there are still several questions regarding its implementation and the data structures involved, in order for this mechanism to function properly. Thus, this mechanism is worth taking a look at.

There are three major data structures involved in Tree RCU: `rcu_data`, `rcu_node`, and `rcu_state`.

Suppose that a CPU registers a callback that will eventually be invoked. Of course, Tree RCU needs to store some information regarding this callback. For this, the implementation maintains some per-CPU data organized in `rcu_data` structures. These structures are per-CPU (see Section 5.2.2), including, among others:

- The last completed grace period number this CPU has seen; used for grace-period ending detection (`completed`).
- The highest grace period number this CPU is aware of having started (`gpnum`).
- A `bool` variable indicating whether this CPU has passed through a quiescent state for this grace period.
- A pointer to this CPU's leaf of hierarchy.
- The mask that will be applied to the leaf's mask (`grpmask`).
- Variables related to callback handling, including this CPU's callback list.
- Variables related to the dynticks interface, as well as a pointer to this CPU's `rcu_dynticks` structure (will be explained later on).
- Variables related to statistics.
- Variables related to CPU hot plugs.

- A pointer to the RCU global state structure (`rcu_state`).

It becomes apparent that when a CPU registers a callback, it is stored in the respective per-CPU data structure. Note that, in Tree RCU, `synchronize_rcu()` is implemented on top of `call_rcu()` (see Section 5.1.3).

Now, as we have already mentioned, when a CPU passes through a quiescent state, it needs to report it to RCU by clearing its bit in the respective leaf node. The hierarchy consists of `rcu_node` structures which include, among others:

- A lock protecting the respective node.
- The current grace period number for this node.
- The number of the last completed grace period for this node.
- A bitmask indicating CPUs or groups that need to check in in order for this grace period to proceed (`qsmask`). In leaf nodes, each bit corresponds to an `rcu_data` structure, and in inner nodes, each bit corresponds to a child `rcu_node` structure.
- A bitmask that will be applied to parent node's mask (`grpmask`).
- The number of the lowest and the highest CPU or group for this node.
- The level of this node in the hierarchy.
- A pointer to the node's parent.

Lastly, the RCU global state, as well as the node hierarchy is included in an `rcu_state` structure. The node hierarchy is represented in heap form in a linear array, which is allocated statically at compile time based on the values of `NR_CPUS` and the `Kconfig` options. Note that small systems will have a hierarchy consisting of a single `rcu_node`. This structure contains, among others:

- The node hierarchy.
- An array of pointers to each level of the hierarchy, as well as the number of levels and kids per node in each level.
- A pointer to the per-CPU variable `rcu_data`.
- The current grace-period number.
- The number of last completed grace period.
- A pointer to the grace-period kthread's `task_struct` (the grace-period kthread will be explained in Section 5.1.3).
- The wait queue where the grace-period task waits.
- A variable representing commands for the grace-period kthread (`gp_flags`).
- A lock that coordinates CPU hot plugs and grace periods.
- A pointer to a list of RCU flavours.

As one may have noticed, there are several values that are propagated through these different structures, e.g., the grace period number. The reason for this will become apparent in Section 5.1.3. However, this was not always the case, and it was the discovery of bugs that often led to changes in the source code.

Finally, we have already mentioned that Classic RCU had a sub-optimal dynticks interface, and one of the main reasons for the creation of Tree RCU was to leave sleeping CPUs lie, in order to conserve energy.

Fortunately, Tree RCU is able to avoid awakening low-power-state dynticks-idle CPUs, thus promoting energy conservation. In order for Tree RCU to accomplish this, a fourth data structure is used, namely the per-CPU `rcu_dynticks` structure. This structure contains, among others:

- A counter tracking the irq/process nesting level.
- A counter tracking the NMI nesting level.
- A counter containing an even value for dynticks-idle mode, else containing an odd value.

These counters enable Tree RCU to wait only for CPUs that are not sleeping, and to let sleeping CPUs lie. The way this is achieved will be described in Section 5.1.3.

In Figure 5.3, a more complete version of the node hierarchy described in Section 5.1.1 is presented. The way CPUs interact with the data structures presented in this figure will be discussed in the next section.

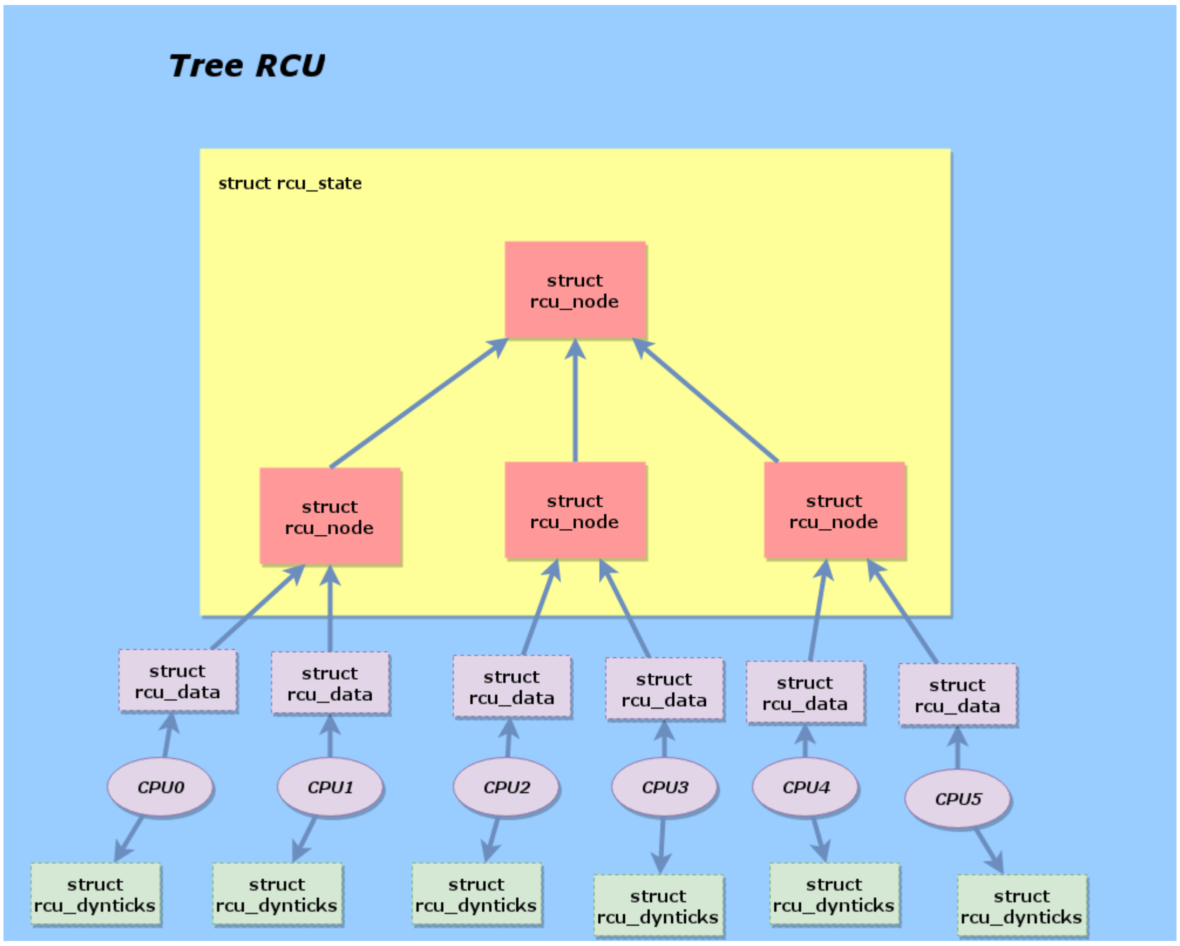


Figure 5.3: Tree RCU node hierarchy – full (adapted from [McKe08a]).

5.1.3 Use Cases

The common usage of RCU involves the registration of a callback, the wait for all pre-existing readers to complete, and finally, the invocation of the callback. During this procedure, special care is taken in order to accommodate sleeping CPUs, offline CPUs and CPU hot plugs, CPUs in userland, and CPUs that fail to report a quiescent state to RCU within a reasonable amount of time. Furthermore, there are many functions and data structures involved throughout this whole procedure (see Section 5.1.2). In the next subsections, we will discuss some use cases of RCU[2], as well as the interaction of RCU with the data structures and the functions involved.

Register a Callback

A CPU can register a callback by invoking `call_rcu()`. This function queues an RCU callback that will be invoked after a specified grace period. The callback is placed in the respective CPU's `rcu_data` structure. This structure contains a list with the callbacks of the respective CPU. The callback list is partitioned in four segments:

1. The first segment contains entries that are ready to be invoked (`DONE` segment).

2. The second segment contains entries that are waiting for the current grace period (`WAIT` segment).

3. The third segment contains entries that are known to have arrived before the current grace period ended (`NEXT_READY` segment).

4. The fourth segment contains entries that *might* have arrived after the current grace period ended (`NEXT` segment).

When a new callback is added to the list, it is inserted at the end of the fourth segment.

In older kernels (e.g., v2.6.x), `call_rcu()` could start a new grace period (see next use case), but this is no longer the case. In newer kernels, the only way a grace period can start directly by `call_rcu()` is if there are too many callbacks queued and no grace period is in progress. Otherwise, a grace period will start from softirq context. But what does this mean exactly?

In Section 2.4.2 we mentioned that every softirq is associated with a function that will be invoked when this type of softirqs is executed (see Listing 2.8). For Tree RCU, this function is called `rcu_process_callbacks()`. So, when an RCU softirq is raised, this function will eventually be invoked (either at the exit from an interrupt handler or from a `ksoftirq/n` kthread), and will start a grace period if there is need for one (e.g., if there is no grace period in progress and the CPU has newly registered callbacks, or there are callbacks that require an additional grace period). RCU softirqs are raised from `rcu_check_callbacks()` which is invoked from scheduling-clock interrupts (more specifically, the `update_process_times()` function); if there is RCU-related work (e.g., if this CPU needs a new grace period), `rcu_check_callbacks()` raises a softirq.

Finally, note that, in Tree RCU, `synchronize_rcu()` is implemented on top of `call_rcu()`. More specifically, `synchronize_rcu()` registers a callback that will awake the caller after a grace period has elapsed. The caller waits on a completion variable, and is consequently put on a wait queue.

[2] These use cases apply to both RCU-sched and RCU-bh.

Start a Grace Period

The function `rcu_start_gp()` is responsible for starting a new grace period. As we have already explained, this function is invoked from softirq context and the `rcu_process_callbacks()` function.

However, in newer kernels, `rcu_start_gp()` neither directly starts a new grace period nor initializes the necessary data structures. It rather advances the CPU's callbacks (i.e. properly re-arranges the segments), and then sets the appropriate flag at the respective `rcu_state` structure to indicate that a CPU requires a new grace period. The grace-period kthread is the one that will initialize the node hierarchy and the `rcu_state` structure, and by extension start the new grace period. Note that there are two grace-period kthreads: one for RCU-sched and one for RCU-bh.

The RCU grace-period kthread excludes concurrent CPU-hotplug operations and then sets the quiescent-state-needed bits in all the `rcu_node` structures in the hierarchy corresponding to online CPUs. It also copies the grace period number and the number of the last completed grace period in all the `rcu_node` structures of the hierarchy (see next use cases). Concurrent CPU accesses will check only the leaves of the hierarchy, and other CPUs may or may not see their respective node initialized. But note that each CPU has to enter the RCU core to acknowledge that a grace period has started. This means that each CPU (except for the one on which the grace-period kthread runs) needs to enter softirq context in order to see the new grace period beginning (via the `rcu_process_callbacks()` function).

The grace-period kthread resolved many races present in older kernels, where when a CPU required a new grace period, it tried to directly initialize the node hierarchy, something that can potentially (through complex scenarios) lead to bugs (see Section 5.3.1).

Pass Through a Quiescent State

In Section 3.2.2 we presented the quiescent states for Tree RCU (RCU-sched), which are:

- Context switch,
- Idle mode (idle loop or dynticks-idle), and
- User-mode execution

When a CPU passes through a quiescent state, it updates its `rcu_data` structure by invoking `rcu_sched_qs()` (or `rcu_qsctr_inc()` for older kernels). This function is invoked from scheduling-related functions (and the scheduler itself), from `rcu_check_callbacks()` (which is called from scheduling-clock interrupts) and from the `ksoftirq/n` kthreads.

But the fact that a CPU has passed through a quiescent state does not mean that RCU knows about it. After all, this fact has been recorded in the respective per-CPU data structure, and not in the node hierarchy. So, a CPU has to report to RCU that it has passed through a quiescent state, and this will happen -again- from softirq context, via `rcu_process_callbacks()`. This procedure is explained in the next use case.

Report a Quiescent State to RCU

After a CPU has passed through a quiescent state, it has to report it to RCU. This is done from the `rcu_process_callbacks()` function which runs in softirq context. This function has a lot of duties, which include:

- Awakening the RCU grace-period kthread via the `rcu_start_gp()` function, in order for it to initialize and start a new grace period, if there is need for one.

- Acknowledging that a new grace period has started/ended. As we have briefly explained, every CPU except for the one on which the RCU grace-period kthread runs has to enter the RCU core and see that a new grace period has started/ended. This is done via the `rcu_check_quiescent_state()` function, which in turn invokes the `note_gp_changes()` function. The latter advances this CPU's callbacks accordingly and records to the respective `rcu_data` structure all the necessary information regarding the grace-period beginning/ending.

- Reporting that the current CPU has passed through a quiescent state (via function `rcu_check_quiescent_state()` which invokes `rcu_report_qs_rdp()` – `cpu_quiet()` for older kernels). A high-level explanation for this procedure was given in Section 5.1.1. If the current CPU is the last one to report a quiescent state, the RCU grace-period kthread is awakened once again, in order for it to clean up after the old grace period and propagate the new `->completed` value to the `rcu_node` structures of the hierarchy.

- Invoking any callbacks whose grace period has ended, via `rcu_do_batch()`.

As can be seen, the RCU grace-period kthread is used heavily to coordinate grace-period endings and beginnings. Apart from this, the locks of the nodes in the hierarchy are used in order to protect concurrent accesses which might lead to problems (see Section 5.3).

Enter/Exit Dynticks-Idle Mode

When a CPU enters dynticks-idle mode (Section 2.5.3), `rcu_idle_enter()` is invoked (or `rcu_enter_nohz()` for older kernels). This function decrements a per-CPU nesting variable (`dynticks_nesting`) and increments a per-CPU counter (`dynticks`), both of which are located in the per-CPU `rcu_dynticks` structure. The `dynticks` counter must have an even value when entering dynticks-idle mode.

When a CPU exits dynticks-idle mode, `rcu_idle_exit()` is invoked (`rcu_exit_nohz()` for older kernels). This function increments the `dynticks_nesting` variable and the `dynticks` counter (which now must have an odd value).

However, we know that dynticks-idle mode is a quiescent state for Tree RCU. So the reason these two variables are needed is the fact that they can be sampled by other CPUs. This way, we can safely determine if a CPU is in a quiescent state (if the `dynticks` counter has an even value), or if a CPU has been in a quiescent state at some point during the grace period in progress (if the `dynticks` value changed during the grace period). The sampling process is performed when a CPU has not reported a quiescent state for a long time, and the grace period needs to end (see next use cases). In reality, there are some other variables that need to be sampled as well (e.g., the `dynticks_nmi` variable), however, the sampling procedure will not be discussed here.

Interrupts and Dynticks-Idle Mode

When a CPU enters an interrupt handler, `rcu_irq_enter()` is invoked from `irq_enter()`. This function increments the `dynticks_nesting` variable, and if the prior value was zero (which means that the CPU was in dynticks-idle mode), also increments the `dynticks` counter (which now must have an odd value).

When a CPU exits an interrupt handler, `rcu_irq_exit()` decrements the `dynticks_nesting` variable, and if the new value is zero (which means that the CPU is entering dynticks-idle mode), also increments the `dynticks` counter (which now must have an even value).

It is self-evident that entering an interrupt handler from dynticks-idle mode means exiting the dynticks-idle mode. Conversely, exiting an interrupt handler might mean the entrance into dynticks-idle mode.

Force Quiescent States

If not all CPUs have reported a quiescent state and several jiffies have passed, then the grace-period kthread is awakened (remember that it is put on a wait queue, but there is a timeout) and will try to force quiescent states on CPUs that have yet to report a quiescent state. More specifically, the function `rcu_gp_fqs()` is called. This function works in two phases:

- In the first phase (`RCU_SAVE_DYNTICK`), snapshots of all CPUs' `dynticks` counters are collected via the `dyntick_save_progress_counter()` function, which in turn is invoked from the `force_qs_rnp()` function. The former samples the specified CPUs' counters, in order to credit them with an implicit quiescent state. In case a CPU is in dynticks-idle mode this function returns 1, in order for `force_qs_rnp()` to call `rcu_report_qs_rnp()`, which reports a quiescent state to RCU.

- In the second phase (`RCU_FORCE_QS`), CPUs that have yet to report a quiescent state are scanned again via the `rcu_implicit_dynticks_qs()` function, which again is invoked from the `force_qs_rnp()` function. The `rcu_implicit_dynticks_qs()` function checks the counters in the per-CPU `rcu_dynticks` structures in order to determine if a CPU has passed through a quiescent state. If a CPU has passed through a quiescent state, or is now in dynticks-idle mode, this function returns 1, in order for `force_qs_rnp()` to call `rcu_report_qs_rnp()`. If there are still CPUs that have not checked in, they are forced into the scheduler in order for them to report a quiescent state to RCU.

5.2 Kernel Environment Modeling

Following a similar pattern as in Section 4.2, in this section we will present the way we scaffolded a non-preemptible Linux-kernel SMP environment. As we will see, the only changes in Tree RCU's source code involve the replacement of per-CPU variables with arrays; the rest of the source code remains untouched.

5.2.1 CPU, Interrupts & Scheduling

CPU

First of all, in order to emulate an SMP system, there is a lock (`pthread_mutex`) for each CPU that, when held, the corresponding thread is running.

As in the verification of Tiny RCU, all CPUs are assumed to be online, CPU hot plugs are excluded, and `CONFIG_NO_HZ_FULL=n` (see Section 2.5.3). All CPUs start out idle, and when a thread wishes to acquire a CPU it invokes `fake_acquire_cpu()`, whose code is shown in Listing 5.1.

```
void fake_acquire_cpu(int cpu)
{
        if (pthread_mutex_lock(&cpu_lock[cpu]))
                exit(-1);
        rcu_idle_exit();
}
```

Listing 5.1: Fake CPU acquisition for Tree RCU.

It is assumed that the CPU of which the lock we are trying to acquire is idle, therefore `rcu_idle_exit()` is called. Of course, if the specified CPU is not idle, one can just acquire the CPU's lock.

Accordingly, when a thread finishes its execution (or wants to release a CPU which will then enter dynticks-idle mode), `fake_release_cpu()` is invoked; see Listing 5.2.

```
void fake_release_cpu(int cpu)
{
        rcu_idle_enter();
        if (pthread_mutex_unlock(&cpu_lock[cpu]))
                exit(-1);
}
```

Listing 5.2: Fake CPU release for Tree RCU.

The number of CPUs in the system can be specified by predefining `CONFIG_NR_CPUS=x` when compiling a test case, where `x` is the number of desired CPUs.

But what about the per-CPU variables present in the Linux kernel? Creating a per-CPU variable means that each CPU will have its own copy of this variable, but a CPU is able to read another CPU's copy (or acquire a pointer to that copy).

In Linux kernel, these variables are created using some compiler/linker directives along with carefully written preprocessor code. Should one want to define a per-CPU variable, special macros (defined in `asm-generic/percpu.h`) must be used. However, since per-CPU variables require significant runtime support (in kernel there are init functions that load the per-CPU sections multiple times – one per CPU), we used arrays to emulate the per-CPU variables. Of course, this led to several, albeit insignificant, source-code changes in all files using per-CPU variables.

Finally, since a thread needs to have knowledge regarding the CPU it is currently running, we implemented two macros, namely `set_cpu()` and `get_cpu()`, which manipulate a thread-local variable indicating the CPU on which a thread runs. This way, a thread is able to get the processor it runs on; but this has to be manually set (via `set_cpu()`) beforehand.

Interrupts & Softirqs

Next in line is the emulation of interrupts and softirqs. As we will explain in the next paragraphs, there are a couple (and possibly more) of ways to emulate interrupts and softirqs. One (and the one we used, ultimately) is to have a per-CPU lock for interrupts and softirqs, which the thread servicing an interrupt holds across the interrupt handler. Of course, the same lock must be acquired when the corresponding CPU disables interrupts or softirqs. This is an overapproximation, but it will suffice for our purposes: since we will be dealing with non-preemptible code, there will be no contention for this lock. Another way to emulate interrupts/softirqs would be to have a separate thread for those. However, since this will make the state space significantly larger, we chose the former approach.

In order to handle the `local_irq_depth`, a per-CPU counter is used. This way, the function `local_irq_disable()` sets `local_irq_depth` to 1, `irq_disabled_flags()` returns `!!local_irq_depth` (i.e. the value of `local_irq_depth`), and `local_irq_enable()` sets `local_irq_depth` to 0. Correspondingly, `local_irq_restore()` sets `local_irq_depth` to the value passed in when function `local_irq_save()` was called. However, since there is also the interrupt lock that the threads contend for, this lock has to be acquired only if the prior value of `local_irq_depth` was zero and the new value is non-zero, and vice versa. Again, this approach works only for non-preemptible code, and modifications are needed should one want to extend it for preemptible code.

Then again, RCU relies mostly on scheduling-clock interrupts (since this is where `rcu_check_callbacks()` is called), and not on interrupts in general, so how can time be modeled given that stateless model checking requires test cases to be finite? Well, the short answer is that all the test cases are time-unaware proofs. More specifically, the exact timing of an interrupt is not so important, since, in real-life scenarios, a CPU might run for awhile with interrupts disabled (with `CONFIG_NO_HZ_FULL=y` this is often the case). Furthermore, in many cases, what interests us is not the exact timing that an interrupt occurs, but the implications an interrupt/softirq might have at a certain point, given a concurrency context. And while there are some timing-based warnings and events, these can be disabled or ignored. This is the case for RCU stall warnings, for which the boot parameters `rcu_task_stall_timeout = 0` and `rcu_cpu_stall_suppress = 1` have been set accordingly.

That said, our version of the interrupt handler (`do_IRQ()` function) emulates the scheduling-clock interrupts. Given that, if there are pending RCU softirqs, they are executed when exiting the handler. Note that only RCU-related softirqs are executed (via `rcu_process_callbacks()`, which, in real kernels, is set as the function to run when a RCU softirq is pending by means of `open_softirq()`), and not softirqs in general. So, our version of `do_IRQ()` invokes `rcu_check_callbacks()` and, if there are pending RCU softirqs (raised by `rcu_check_callbacks()`), it invokes `rcu_process_callbacks()`. Of course, we could just call `rcu_process_callbacks()`, but we wanted our modeling of scheduling-clock interrupts to be as close to their real implementation as possible, so we do not invoke `rcu_process_callbcaks()` unconditionally.

Another key point for the interrupt lock is that, in theory, it should not be held when executing softirqs, since these are executed with interrupts enabled. Then again, this should not make much difference since the kernel is supposed to be non-preemptible. Of course, interrupts/softirqs could be approximated with a separate thread, but, as we have already mentioned, this would make the state space significantly larger.

Finally, the kernel is assumed to be compiled with dynticks mode enabled, and one has to acquire the CPU's lock before acquiring the corresponding interrupt lock. In real kernels, an idle thread is employed for each CPU and these idle threads run when no other thread does. Assuming that dynticks mode is enabled (default option), the only purpose of these threads is to conserve power, and when they run, they do the bare minimum. However, since we are not trying to verify properties of the idle threads, there is no point in modeling them and including them in our tests.

Scheduling

As for scheduling functions, `cond_resched()` is modeled by having the running thread drop the CPU's lock and then (possibly) re-acquire it, as in Tiny RCU, but with the function `rcu_note_context_switch()` being invoked before releasing the lock of the incoming CPU.

In reality, the best way to model this function would be to drop the CPU's lock and then acquire the lock of some random CPU, possibly the one just released. However, as we have mentioned numerous times, the test cases to be analyzed must be deterministic in the sense that in a given state, executing a given execution step must always take the system to the same new state. This means that Nidhugg cannot e.g., check the time, or discover bugs that are caused by data non-determinism, so `cond_resched()` practically *has* to acquire the same CPU it released.

But, as we will see Section 5.3, this did not prevent us from reproducing older bugs using Nidhugg, since the code in our test cases aims to be CPU-specific (not thread-specific), and RCU read-side critical sections are not preemptible.

Furthermore, there also is the `resched_cpu()` function. The proper way to model this function would be to force the task holding the specified CPU's lock to execute `cond_resched()`, but at a point in its execution where both interrupts and preemption are enabled.

However, since `resched_cpu()` is used in kthread kicking (used when a CPU has failed to report a quiescent state for a grace period after a specified amount of time) and our tests are time-unaware, this function is stubbed.

5.2.2 Kernel Definitions

In this section we will discuss the way we scaffolded a non-preemptible Linux-kernel SMP environment. Our tests focused on four different kernel versions: v3.19, v2.6.31.1, v2.6.32.1 and v3.0. One can chose the kernel version that will be included in the test case by focusing the preprocessor's attention to the right directory. The reasons these particular versions were chosen will become apparent in Sections 5.3-5.5, but having many kernel versions allows us to run the same test case for multiple configurations.

As in Chapter 4, many definitions were copied from the Linux-kernel. These include data types like `u8`, `u16`, etc, compiler directives like `offsetof()`, macros like `ACCESS_ONCE()`, list data types and functions, memory barriers, and various other kernel primitives.

On the other hand, many primitives had to be replaced or stubbed; we supplied once more empty files for includes, and provided some "cheater" definitions based on restricted `Kconfig` options in file `fake_defs.c`. These include definitions relevant to the CPUs (e.g., `NR_CPUS`), RCU-related definitions that are normally configured at compile time (e.g., `CONFIG_RCU_BOOST`), special compiler directives, tracing functions, etc. Again, the `BUG_ON()` macro and its relatives (e.g., `WARN_ON()`) have been replaced by `assert()` statements. Normally, `BUG_ON()` makes the kernel panic and brings the system down, while `WARN_ON()` writes something to the kernel log (which can be viewed via `dmesg`). In addition, it is important to note that the RCU-bh grace-period kthread has been disabled, but it can be enabled with `-DENABLE_RCU_BH`. Note that the aforementioned do not affect the code's behaviour since we only stubbed primitives irrelevant to our tests (e.g., some primitives related to grace-period expediting) and provided our own definitions for some other primitives in order for them to work with our modeling of the CPU and interrupts.

Memory barriers are provided for x86 (default) and POWER (`-DPOWERPC`) configuration. Unfortunately, not all tests can run under the POWER memory model, since there are primitives like `pthread_mutex_trylock()` that are not supported by Nidhugg for this memory model (also see Section 5.2.3). In cases where architecture-dependent primitives were needed, we included as many different primitives as possible. However, in many cases Nidhugg supported the respective primitive only for x86 architectures (see Section 5.2.3).

5.2.3 Synchronization Mechanisms

In Section 2.6, we discussed some of the synchronization mechanisms the kernel provides. In this section we will discuss how the primitives these mechanisms provide were emulated, as well as some implementation details.

Atomic Operations

While the atomic data type `atomic_t` was copied directly from the Linux kernel, this is not the case for atomic operations like `atomic_read()`, `atomic_set()`, `atomic_add()`, etc., since their implementation is architecture dependent. In order to emulate those, we used some gcc language extensions which are also supported by the clang compiler [GCCA, Clan]. For example, the definition for `atomic_add()` is as follows: `#define atomic_add(i, v) __atomic_add_fetch(&(v)->counter, i, __ATOMIC_RELAXED)`. In summary, all the atomic primitives used in the Linux kernel are substituted by the respective gcc built-ins.

These GCC built-ins are used for *memory model aware* atomic operations; they match the requirements for the C++11 memory model. However, Nidhugg supports these built-ins only for SC, TSO and PSO, and only for the `__ATOMIC_SEQ_CST` model (which enforces total ordering with all other `__ATOMIC_SEQ_CST` operations), even if otherwise specified. This means that they can be used only for the x86 configuration, since x86 CPUs do not reorder atomic operations with stores and loads [Wikia].

Spinlocks & Mutexes

In non-RT kernels, `spinlock_t` is just a wrapper for `raw_spinlock_t`, which is the actual implementation for normal spinlocks. Of course, while `raw_spinlock_t` is architecture independent, it delegates low-level operations to `arch_spinlock_t`, which is architecture dependent. Since we wanted an architecture independent implementation that is supported by Nidhugg, we used `pthread_mutex`es for the emulation of spinlocks. In like manner, `pthread_mutex` has been used in place of `struct mutex`. Since many spinlocks and mutexes are initialized statically in the kernel, the `--disable-mutex-init-requirement` Nidhugg option is required for most tests to run.

Functions related to the runtime locking correctness validator (see Section 2.6.4) have been disabled [McKe10a].

Completions

In Section 2.6.5 we explained how completion variables work in the kernel. For our tests, we kept the definition from Figure 2.10, but wait queues had to be modeled. For this, we tried two different approaches, both of which worked.

The first approach involved the usage of condition variables to model wait queues. So, when a thread waited for an event, it just waited on a condition variable. But since the usage of a condition variable might make the state space larger, and it needs to tolerate things like premature wakeups (although this did not pose a problem), we tried and favored the second approach.

The second approach involved the emulation of completions by simply spin-waiting. Since a thread waiting on a completion is put on a wait queue until a condition is satisfied, we used spin loops in order to emulate wait queues. Nidhugg automatically transforms all spin loops to `__VERIFIER_assume()` statements, where if the condition does not hold, the execution blocks indefinitely [Beye15]. Before waiting on a spin loop, a thread drops the corresponding CPU's lock; it will try to re-acquire it after the condition has been satisfied. Since this is a quiescent state for RCU, `rcu_note_context_switch()` (and possibly `do_IRQ()`, in order for `rcu_process_callbacks()` to be called) could be invoked before the thread releases the CPU's lock. However, if the thread waiting on the completion variable is not the only thread running on the CPU, this is not necessary; these functions can be called from other threads running on the same CPU as well.

5.3 Reproducing Older Bugs

In this section we will try to reproduce some older bugs found in the Linux-kernel, with little knowledge of the circumstances they occur in. The test cases are designed to run in multiple kernel versions, in order to determine if these bugs were actually fixed.

5.3.1 Bug #1: Synchronization issues for `rcu_process_gp_end()`

In Section 5.1.3 we mentioned that the RCU grace-period kthread cleans up after the grace-period ending. However, this was not always the case. In older kernel versions, the RCU grace-period kthread did not exist. Moreover, when a CPU entered the RCU core or invoked `call_rcu()`, it checked for grace-period endings by directly comparing the number of the last completed grace period in the `rcu_state` structure with the number of the last completed grace period in the respective `rcu_data` structure (via the `rcu_process_gp_end()` function). In newer kernels, the `note_gp_changes()` function compares the number of the last completed grace period in the respective `rcu_node` structure with the number of the last completed grace period in the current rcu_data structure, *while holding the node's lock*, that way excluding concurrent operations on this node.

In kernel v2.6.32, commit d09b62dfa336 fixed a bug related to unsychronized accesses to the ->completed counter in the rcu_state structure, which caused the advancement of callbacks whose grace period had not yet expired [LKMLc, LKMLa]. In the next paragraphs, we will show how these unsynchronized accesses led to too-short grace periods.

Our primary goal is to construct a test case that exposes this bug and provides us with a witness. But how can we construct a proper test case that exposes a bug, if the bug is not known a priori? Well, since we know that it is related to rcu_process_gp_end() function, we can take a look at the relevant portion of this function in Listing 5.3. As can be seen from this listing, the access to the ->completed counter is completely unprotected.

```
completed_snap = ACCESS_ONCE(rsp->completed);  /* outside of lock. */

/* Did another grace period end? */
if (rdp->completed != completed_snap) {

        /* Advance callbacks.  No harm if list empty. */
        rdp->nxttail[RCU_DONE_TAIL] = rdp->nxttail[RCU_WAIT_TAIL];
        rdp->nxttail[RCU_WAIT_TAIL] = rdp->nxttail[RCU_NEXT_READY_TAIL];
        rdp->nxttail[RCU_NEXT_READY_TAIL] = rdp->nxttail[RCU_NEXT_TAIL];

        /* Remember that we saw this grace-period completion. */
        rdp->completed = completed_snap;
}
```

Listing 5.3: Unsynchronized accesses to the ->completed counter.

So, the first step was to inject a BUG_ON() statement in the **if**-body to determine if it was possible for a thread to pick up the ->completed value and then use the completed_snap while the ->completed variable has changed. The answer was affirmative.

The next step was to determine if this could potentially lead to a CPU starting a new grace period without having noticed the last grace period's ending. Again, the injection of a BUG_ON() statement which compared the current grace period's number to the number of the grace period whose completion the CPU notices, proved that this is possible.

With these clues, we were able to construct a simple test case (Listing 5.4) which proved that these unsynchronized accesses can lead to too-short grace periods (file gp_end_bug.c). Note that too-short grace periods are not always of concern; however, they could potentially allow an RCU read-side critical section to span a grace period, which breaks the fundamental RCU Grace-Period guarantee. Given that, the test case provided has to have a reader seeing changes happening before the beginning of the grace period *and* after the end of the same grace period within a single RCU read-side critical section.

In Listing 5.4 we can see that there are three threads and two CPUs. The thread_update() thread runs on CPU0 and the thread_reader() thread runs on CPU1. The thread thread_helper() represents a random thread running on CPU0 that can -potentially- occupy the CPU after thread_update() has blocked due to synchronize_rcu(). A sequence of events that exposes the aforementioned bug is presented below:

0. CPU0 has registered a callback, started a grace period, passed through a quiescent state and reported it to RCU.

1. CPU1 sees that there is a grace period in progress (by taking a timer interrupt) and passes through a quiescent state, but does not report it to RCU yet.

2. CPU1 starts an RCU read-side critical section and executes `r_x = x`.
3. CPU1 takes a timer interrupt and enters RCU core (but does not end the grace period just yet).
4. CPU0 executes `x = 1` and then `synchronize_rcu()`. Now `synchronize_rcu()` registers a callback. In v2.6.31.1 and v2.6.32.1, `__call_rcu()` calls `rcu_process_gp_end()`. So CPU0 calls `rcu_process_gp_end()`, but CPU1 has not ended the grace period yet, hence, CPU0 sees that the current grace period is still in progress. Note that CPU1 had already started its RCU read-side critical section when CPU0 executed `synchronize_rcu()`.
5. CPU1 ends the first grace period and updates the `->completed` value. CPU1 does not need a new grace period, so it does not start one.
6. CPU0 checks to see if a new grace period has started (by comparing `rdp->gpnum` with `rsp->gpnum`, but a new grace period has not started. This happens when CPU0 executes `rcu_check_for_new_grace_period()`.
7. CPU0 has registered callbacks and sees that no grace period is in progress (by seeing that `rcu_gp_in_progress() == 0`), starts a new grace period, and advances its callbacks from `NEXT` to `WAIT` segment. CPU0 has still not seen the previous grace-period ending.
8. CPU0 takes another timer interrupt and enters RCU core. It sees that a grace period has completed (via `rcu_process_gp_end()`) and advances its callbacks from `WAIT` to `DONE`. That means that the callback corresponding to `synchronize_rcu()` is ready to invoke, and CPU0 invokes it during the same interrupt.
9. CPU0 executes `y = 1`.
10. CPU1 executes `r_y = 1`.

From the aforementioned, we conclude that the outcome

```
r_x == 0 && r_y == 1
```

is possible, something that violates the RCU Grace-Period guarantee, since the time between registration and invocation of a callback needs to span a grace period (and thus a full set of quiescent states), and this is not the case here.

Finally, some notes regarding this bug:

- This bug existed in both single-node and multi-level hierarchies (can be constructed with `-DCONFIG_RCU_FANOUT=x`, although a slightly different test from one presented in Listing 5.4 is needed), since it does not rely on interactions with the node hierarchy.
- This bug is not present in Linux kernel v3.0 (`-DKERNEL_VERSION_3` is needed in order for the test to run in v3.0), which means that it was indeed fixed. The reason for that is that `rcu_start_gp()` in v3.0 calls `__rcu_process_gp_end()`, which guarantees that a CPU will see a grace-period ending before a grace-period beginning, something that does not happen in v2.6.32.1. However, it was present in previous versions as well, e.g., 2.6.31.1.
- Only two CPUs are required to provoke the bug, and only one of them has to invoke `call_rcu()`.
- Only one grace period is required to provoke the bug, meaning that it does not rely on CPUs being unaware of grace period endings and beginnings (e.g., when a CPU is in dynticks-idle mode). Of course, the latter implies that `CONFIG_NO_HZ=y/n` should not affect the test results.

- `force_quiescent_state()` is not required to provoke the bug, although frequent calls to this function would expose it easier in real-life scenarios.
- It is not caused by weak memory ordering; the test runs under sequential consistency.
- Nidhugg produced the aforementioned sequence of events in only $0.56s$ (compilation and Nidhugg transformation time included), and used 30.85 MB of memory (this includes the memory used for RCU data structures).

5.3.2 Bug #2: Bug between grace-period forcing and initialization

In this section we will demonstrate that a previously reported bug is not in reality a bug using Nidhugg. More specifically, we will deal with Linux kernel v2.6.31 and an alleged long-grace-period race between grace-period forcing and initialization [LKMLb]. This bug was supposed to be fixed with commit 83f5b01ffbba; however, it is highly possible that the Bug #1 in the previous section was the bug hiding behind the too-short grace periods observed, since these two bugs were related [LKMLa, McKe09a].

The commit log for this bug states that very long RCU read-side critical sections can cause a race between `force_quiescent_state()` and `rcu_start_gp()` as follows:

1. CPU 0 calls `force_quiescent_state()`, sees that there is a grace period in progress, and acquires `->fsqlock`.
2. CPU 1 detects the end of the grace period, and so `cpu_quiet_msk_finish()`[3] sets `rsp->completed` to `rsp->gpnum`. This operation is carried out under the root `rnp->lock`, but CPU 0 has not yet acquired that lock. Note that `rsp->signaled` is still `RCU_SAVE_DYNTICK` from the last grace period.
3. CPU 1 calls `rcu_start_gp()`, but no one wants a new grace period, so it drops the root `rnp->lock` and returns.
4. CPU 0 acquires the root `rnp->lock` and picks up `rsp->completed` and `rsp->signaled`, then drops `rnp->lock`. It then enters the `RCU_SAVE_DYNTICK` leg of the switch statement.
5. CPU 2 invokes `call_rcu()`, and now needs a new grace period. It calls `rcu_start_gp()`, which acquires the root `rnp->lock`, sets `rsp->signaled` to `RCU_GP_INIT` (too bad that CPU 0 is already in the `RCU_SAVE_DYNTICK` leg of the switch statement!) and starts initializing the `rcu_node` hierarchy. If there are multiple levels to the hierarchy, it will drop the root `rnp->lock` and initialize the lower levels of the hierarchy.
6. CPU 0 notes that `rsp->completed` has not changed, which permits both CPU 2 and CPU 0 to try recording it concurrently. If CPU 0's update prevails, later calls to `force_quiescent_state()` can count old quiescent states against the new grace period, which can in turn result in premature ending of grace periods.

Note that this bug is supposed to be present in kernel builds with multi-level `rcu_node` hierarchies.

However, trying to convert this failure scenario into a test case, too-short grace periods did not occur. This was admittedly odd at first sight, but looking at `force_quiescent_state()` in Listing 5.5 states otherwise. As can be seen, CPU0 cannot enter the `RCU_SAVE_DYNTICK` leg of the switch statement since, after picking up the values for `rsp->completed` and `rsp->signaled`, the `lastcomp == rsp->gpnum` condition should hold (no grace period is in progress), and CPU0 will go to the `unlock_ret` label. That certainly confirmed our test results, but since there were reported failures at the time, we might as well be missing something here.

[3] In older kernel versions this function was embedded in `cpu_quiet_msk()`.

So what if the commit log was wrong? Maybe the sequence of events was not correct, but the bug was still there. Consequently, the next step is to consider a slightly different sequence of events that may provide us with a witness:

1. CPU 0 calls `force_quiescent_state()`, sees that there is a grace period in progress, and acquires `->fsqlock`.

   + It also acquires the root `rnp->lock` and picks up `rsp->completed` and `rsp->signaled`, and then drops `rnp->lock`.

2. CPU 1 detects the end of the grace period, and so `cpu_quiet_msk_finish()` sets `rsp->completed` to `rsp->gpnum`. This operation is carried out under the root `rnp->lock`, but CPU 0 has not yet acquired that lock. Note that `rsp->signaled` is still `RCU_SAVE_DYNTICK` from the last grace period.

3. CPU 1 calls `rcu_start_gp()`, but no one wants a new grace period, so it drops the root `rnp->lock` and returns.

4. CPU 0 enters the `RCU_SAVE_DYNTICK` leg of the switch statement.

5. CPU 2 invokes `call_rcu()` and now needs a new grace period. It calls `rcu_start_gp()`, which acquires the root `rnp->lock`, sets `rsp->signaled` to `RCU_GP_INIT` (too bad that CPU 0 is already in the `RCU_SAVE_DYNTICK` leg of the switch statement!) and starts initializing the `rcu_node` hierarchy. If there are multiple levels to the hierarchy, CPU 2 will drop the root `rnp->lock` and initialize the lower levels of the hierarchy.

6. CPU 0 notes that `rsp->completed` has not changed, which permits both CPU 2 and CPU 0 to try updating it concurrently. If CPU 0's update prevails, later calls to `force_quiescent_state()` can count old quiescent states against the new grace period, which can in turn result in premature ending of grace periods.

The aforementioned sequence of events also cannot cause the counting of old quiescent states against a new grace period. If CPU0 picks up the `->completed` value before the grace period ends, it will not try to update the `->dynticks_completed` value in step 6, because the `->completed` value is different from the one it picked up.

In effect, it seems that there is no such race between grace-period forcing and initialization. In order to prove this, we provide some tests of the basic premise: it should be impossible for a CPU trying to force a quiescent state to pick up the `->completed` and `->signaled` values *before* a grace period ends and then update the `->dynticks_completed` value *after* a new grace period has started. More specifically, it is impossible to enter the `RCU_SAVE_DYNTICK` leg of the switch statement in `force_quiescent_state()` before a grace period ends and record the `dynticks_completed` value after a new grace period has started. The results we got confirmed this, along with the fact that it is impossible for CPU0 to be in the `RCU_SAVE_DYNTICK` leg of the switch statement, while `rsp->signaled == RCU_GP_INIT` and `lastcomp == rsp->gpnum`.

Since this alleged bug needs (at least) three CPUs to be provoked, `-DCONFIG_NR_CPUS=3` is required in our tests. In addition, `-DCONFIG_RCU_FANOUT=2` is needed in order to create a multi-level hierarchy. Lastly, `-DFQS_NO_BUG` is required in order for all the necessary assertions to be inserted.

5.4 Validating Tree RCU's Publish-Subscribe Guarantee

In this section we will mechanically validate the Publish-Subscribe guarantee of Tree RCU, Linux-kernel v3.19. Since we have fully explained and analyzed this guarantee in Section 3.2.1, we will not go into details in this section. The code fragment in Listing 5.6 constitutes a test for this guarantee.

This test is configured to run under both TSO (x86) and POWER memory model. For POWER, `-DPOWERPC` has to be passed to the preprocessor so that the correct memory barriers from `fake_defs.h` file are included. When `-DORDERING_BUG` is passed to the preprocessor `rcu_assign_pointer()` is not used, something that can lead to bugs, assuming a memory model that reorders stores. A simple `BUG_ON()` statement at `use_gp()` function is used to check whether the reader can see uninitialized values. For both TSO and POWER, Nidhugg is able to validate the Publish-Subscribe guarantee very quickly: two interleavings in total are explored and the process requires about 0.08 seconds (compilation and Nidhugg transformation time included).

As expected, `rcu_assign_pointer()` was absolutely necessary for POWER, whereas the memory model on x86 did not really require the primitive (something that is expected considering the fact that x86 does not reorder stores). That said, `rcu_assign_pointer()` in x86 architectures (except probably the Pentium Pro family of processors) boils down to a compiler barrier. Of course, aggressive compiler optimizations would affect the outcome as well, and `rcu_assign_pointer()` has to be used in order to prevent such optimizations, but Nidhugg cannot find a bug in such cases, since the bug is unrelated to scheduling and the underlying memory model.

Finally, this test is performed from the publisher's point of view. Performing an analogous test from the subscriber's point of view would require support for a memory model that reorders dependent loads (like the memory model in DEC Alpha architecture).

5.5 Validating Tree RCU's Grace-Period Guarantee

In this section we will mechanically validate the Grace-Period guarantee of Tree RCU for a non-preemptible Linux-kernel environment (v3.19). Since we have already provided an in-depth explanation for this guarantee in Section 3.4.1, in Listing 5.7 we present the structure for a canonical litmus test for this guarantee. However, before we present the results of this test, we will briefly discuss our modeling of the Linux-kernel architecture. In Listing 5.7 the result:

$$\texttt{r\_x == 0 \&\& r\_y == 1} \tag{5.1}$$

violates the Grace-Period guarantee of RCU.

First of all, we have a system with two cores, represented by two mutexes respectively. We also have three basic threads: the updater, the reader and the RCU grace-period kthread. The RCU-bh grace period kthread is disabled in order to reduce the state space, but it can be enabled with `-DENABLE_RCU_BH`. We can assume that the updater and the RCU grace-period kthread run on the same CPU (e.g., CPU0), and that the reader runs on the other CPU (e.g., CPU1). Different configurations were tried as well and they did not affect the outcome; this particular configuration was chosen because the updater and the RCU grace-period kthread take advantage of each other's context switches. Note that we could have ignored the RCU grace-period kthread and invoke `rcu_gp_init()` and `rcu_gp_cleanup()` directly, in order to reduce the state space; however, this is an approximation and not the way the real kernel works, and this is why we kept the RCU grace-period kthread in our tests. For RCU initialization, `rcu_init()` function is called. Since there are only two CPUs in our modeling, a single-node hierarchy is created. All CPUs start out idle (`rcu_idle_enter()` is called for each CPU), and `rcu_spawn_gp_kthread()` is called in order to spawn the RCU grace-period kthread.

Of course, interrupt context had to be emulated as well. For this, there are calls to `do_IRQ()` in various points of the test code. For example, we did try to insert a call to `do_IRQ()` right after the updater registers a callback and busy-waits on the completion variable (i.e. after calling `synchronize_rcu()`). Although this is not the case in the test code anymore, this invocation represented an interrupt that occured *after* the updater had registered its callback. In general, even though we do not care about the exact timing of interrupts, it is the occurence of an interrupt within a specific context that causes a grace period to advance.

Thus, calls to `do_IRQ()` have been inserted into places that enable the advancement of a grace period. This may not always be the case (i.e. a grace period may not end for some explored executions), but in fact we want to enable both of these scenarios to happen. A set of per-CPU locks for interrupts (and for interrupt disabling) and the approximation of interrupts with a separate thread might also be a good approach (see Section 5.2), but this would be very tough on the state space.

After running the test (with an unroll value in order for the test to be finite), Nidhugg states that verification is successful. This requires approximately 17 minutes (1051.83 seconds) under SC; see the first line of Table 5.1. Unfortunately, this test runs only under SC and TSO since Nidhugg does not have full support for the POWER memory model (see Section 5.2.2). But can we trust the result we got? After all there might be a bug in our scaffolding of the Linux-kernel's environment, or there might be a bug in Nidhugg.

In order to strengthen our result, we injected some bugs in the test and the RCU source code (for the respective modifications in Tree RCU's source code see Appendix A). There are two kinds of injected bugs:

1. Bugs that make the grace period too-short, thus permitting an RCU read-side critical section to span the grace period.

2. Bugs that prevent the grace period from completing.

Both of these bug types represent RCU failures. The first type results in failed verification, since Assertion 5.1 is violated. The second type has to be used with an `assert(0)` statement after `synchronize_rcu()` in order to demonstrate that the grace period does not end, and results in successful verification.

Below, we present a list of defines that enable these test cases scenarios, along with an explanation of each injection and the outcome of the verification:

`-DASSERT_0`: An `assert(0)` statement is inserted after the `synchronize_rcu()` function. This result in failed verification, as expected. What this assertion does, however, is that it shows that the grace period *can* end, and that there are some explored executions in which it does; in other words, it provides liveness guarantees. We will use this injection in conjunction with some of the next bug injections in order to determine whether the grace period ends or not.

`-DFORCE_FAILURE_1`: This injection forces the reader to pass through and report a quiescent state during its read-side critical section. Of course, this is not permitted (since quiescent states cannot occur during RCU read-side critical sections) and, as expected, results in failed verification in a way similar to the one for Tiny RCU (Section 4.3).

`-DFORCE_FAILURE_2`: A **`return`** statement is placed at the beginning of the `synchronize_rcu()` function so that it returns immediately. Of course, this results in failed verification since the updater does not wait for pre-existing readers to complete their RCU read-side critical sections, and RCU read-side critical sections are not permitted to span a grace period.

`-DFORCE_FAILURE_3`: This injection makes the `rcu_gp_init()` function clear the `->qsmask` variables instead of setting them appropriately. The `rcu_gp_init()` function is invoked from the RCU grace-period kthread at the beginning of each grace period and its purpose is to initialize the grace period. Of course, since the `->qsmask` variables are cleared from the beginning of the grace period, the grace period can end immediately; in other words, the grace-period kthread does not wait for pre-existing readers to complete (this can be considered a more complex variant of injection #2 above). As expected, this injection results in failed verification.

-DFORCE_FAILURE_4: In this injection the `rcu_gp_fqs()` function is made to clear the `->qsmask` variables instead of waiting for the CPUs to clear their respective bits. Of course, in order for `rcu_gp_fqs()` to clear the `->qsmask` variables, the respective CPUs (in our case, the reader) has to be in dynticks-idle mode (or the CPU must have passed through a quiescent state at some point – remember that the respective dynticks counters are sampled). Consequently, in our code, CPU0 calls the `rcu_gp_fqs()` function, and CPU1 enters and exits dynticks-idle mode within its RCU read-side critical section, which enables CPU0 to prematurely end the grace period. This can be considered an even more complex variant of injection #2, and results in failed verification, as expected.

-DFORCE_FAILURE_5: This injection makes the `__note_gp_changes()` function to clear the respective `rnp->qsmask` bit for this CPU (`rnp->qsmask &= ~rdp->grpmask`). This function is called when a CPU enters RCU core (i.e. executes the `rcu_process_callbacks()` function) in order to record the beginnings and ends of grace periods. However, instead of just recording a grace period beginning, `__note_gp_changes()` is now made to also clear the `rnp->qsmask` bit, which implies that this CPU reported a quiescent state for the new grace period. As expected, this injection results in failed verification.

-DFORCE_FAILURE_6: Essentially, what this injection does is delete the **`if`** statement checking for zero `->qsmask` and calling the function `rcu_preempt_blocked_readers_cgp()`, in the `rcu_report_qs_rnp()` function. This **`if`** statement just checks whether the bitmask for this node is cleared (i.e. the CPU reporting a quiescent state is the last to check in for its group), in order to acquire the parent node's lock. In a real kernel, this should result in too-short grace periods, since a signal that will prematurely awake the grace-period kthread is sent, at least assuming that there are multiple CPUs represented. In our case, however, it does not lead to too-short grace periods, as we will explain in the next paragraphs.

If this **`if`** is deleted, and since `rnp->parent == NULL` holds, `rcu_report_qs_rsp()` is called. Now `wait_event_interruptible_timeout()` just busy-waits on a condition. So, even with `rcu_report_qs_rsp()` called, until both CPUs check in, it is impossible for the RCU grace-period kthread to get past `wait_event_interruptible_timeout()` and end the grace period, since the bits of both processors are not cleared. Note that, in our modeling, `wake_up()` boils down to a no-op – there is no need to wake up someone who is just spinning.

However, if we were dealing with a two-level tree, the caller of `rcu_report_qs_rnp()` would move up one level and thus trigger a `WARN_ON_ONCE()` statement that checks whether the child node's bits are cleared. Hence, this test automatically sets the number of CPUs to `CONFIG_RCU_FANOUT_LEAF + 1` (i.e. to 17, since the default value of `CONFIG_RCU_FANOUT_LEAF` is 16 in kernel v3.19), and uses 19 as an unroll value. This is because there are some loops that need to be unrolled at least as many times as the number of CPUs that the test uses plus one. With these preconditions, this injection leads in failed verification.

-DLIVENESS_CHECK_1: This injection sets `rdp->qs_pending` to zero in the `__note_gp_changes()` function. The `__note_gp_changes()` function updates CPU-local `rcu_data` structures. So, since `rdp->qs_pending` is set to zero, there is no need for a CPU to report a quiescent state to RCU, and the CPUs do not bother reporting a quiescent state. Of course, this injection should prevent grace periods from completing. When it is used in conjunction with `-DASSERT_0`, it results in successful verification. This means that no assertion is violated, which in turns means that the `assert(0)` statement after `synchronize_rcu()` is not reached, which means that a grace period does not end.

`-DLIVENESS_CHECK_2`: A **`return`** statement is placed at the beginning of the `rcu_sched_qs()` function so that it returns immediately. In effect, this means that CPUs cannot record in their `rcu_data` structures their passing through a quiescent state. As expected, this should also prevent grace periods from completing, and needs to be used in conjunction with `-DASSERT_0`. In a similar manner with the previous injection, it results in successful verification.

`-DLIVENESS_CHECK_3`: A **`return`** statement is placed at the beginning of the `rcu_report_qs_rnp()` function so that it returns immediately. This means that CPUs cannot report their passing through a quiescent state to RCU, which in turn means that a grace period cannot complete. This injection also needs to be used in conjunction with `-DASSERT_0`, and results in successful verification as well.

For the `FORCE_FAILURE_6` test, an unroll value `unroll=19` has been used and `CONFIG_NR_CPUS` has been set appropriately, while for all the other tests an unroll value `unroll=5` has been used. All experiments have been performed on a 64-bit Debian Linux machine (kernel version: 3.16.0-4-amd64) with an Intel E8400 processor (two cores) and 2GB of RAM. As can be seen above and in Table 5.1, all tests have the desired outcome, something that makes our results significantly stronger.

Preprocessor Options	*Expected*	*Nidhugg*	*SC*			*TSO*		
			Time	*E/B*	*Memory*	*Time*	*E/B*	*Memory*
-	Success	Success	1051.83	24 740/20	64.34	1130.94	24 740/20	64.84
`-DASSERT_0`	Failure	Failure	2.08	31/6	34.42	2.29	31/6	34.40
`-DFORCE_FAILURE_1`	Failure	Failure	2.20	35/6	34.43	2.55	35/6	34.37
`-DFORCE_FAILURE_2`	Failure	Failure	0.80	3/0	34.24	0.89	3/0	34.28
`-DFORCE_FAILURE_3`	Failure	Failure	567.86	13 256/8	47.92	612.75	13 256/8	49.74
`-DFORCE_FAILURE_4`	Failure	Failure	4.40	73/6	34.50	4.97	73/6	34.54
`-DFORCE_FAILURE_5`	Failure	Failure	1.34	8/1	34.62	1.46	8/1	34.86
`-DFORCE_FAILURE_6`	Failure	Failure	4.58	2/0	109.75	4.93	2/0	109.87
`-DLIVENESS_CHECK_1 -DASSERT_0`	Success	Success	15.62	588/20	34.38	17.08	588/20	34.29
`-DLIVENESS_CHECK_2 -DASSERT_0`	Success	Success	15.52	588/20	34.82	17.09	588/20	34.82
`-DLIVENESS_CHECK_3 -DASSERT_0`	Success	Success	17.70	668/20	34.52	21.26	668/20	34.68

Table 5.1: Results for Tree RCU litmus test (time in seconds, memory in MB).

In Table 5.1, the second column indicates the expected outcome, and the third column indicates the actual outcome from Nidhugg. The "Time" columns present the total wall-clock time in seconds (compilation and Nidhugg transformation time are included). The columns labeled E/B present the number of explored and sleep-set blocked interleavings (i.e. interleavings that were only partially and unnecessarily explored; their exploration could be avoided by a better DPOR algorithm such as e.g., optimal DPOR [Abdu14]) under SC and TSO respectively.

As also can be seen from Table 5.1 there is little to no overhead when going from SC to TSO, something expected when using chronological traces [Abdu15]. What is interesting though, is the fact that the total number of explored executions is the same for both SC and TSO. The reason for that is that there are a lot of memory fences in the source code of Tree RCU which prevent possible reorderings from happening. But even if reorderings were possible, these bug injections do not rely on the employed memory model but instead violate the assertions algorithmically.

Finally, as expected, the memory consumption in our tests is very low given the size of the source code tested. In addition, we can see that the overhead when going from SC to

TSO is practically zero. This is due to the fact that the number of explored interleavings in SC and TSO is exactly the same. In the `FORCE_FAILURE_6` bug injection the memory consumption is increased due to the higher unroll value (`unroll=19`) and the increased number of RCU data structures (`CONFIG_NR_CPUS=17`).

```
void *thread_update(void *arg)
{
        set_cpu(cpu0);
        fake_acquire_cpu(get_cpu());

        call_rcu(&cb1, dummy);
        cond_resched();
        do_IRQ();

        x = 1;
        synchronize_rcu();
        y = 1;

        fake_release_cpu(get_cpu());
        return NULL;
}

void *thread_helper(void *arg)
{
        set_cpu(cpu0);
        fake_acquire_cpu(get_cpu());

        do_IRQ();

        fake_release_cpu(get_cpu());
        return NULL;
}

void *thread_reader(void *arg)
{
        set_cpu(cpu1);
        fake_acquire_cpu(get_cpu());

        do_IRQ();
        cond_resched();

        rcu_read_lock();
        r_x = x;
        do_IRQ();
        r_y = y;
        rcu_read_unlock();

        fake_release_cpu(get_cpu());
        return NULL;
}

int main()
{
        /* ... */
        /* Initializations and creation of threads */
        /* ... */

        BUG_ON(r_x == 0 && r_y == 1);

        return 0;
}
```

Listing 5.4: Too-short grace period caused by synchronization issues in `rcu_process_gp_end()`.

```
	if (ACCESS_ONCE(rsp->completed) == ACCESS_ONCE(rsp->gpnum))
		return;	/* No grace period in progress, nothing to force. */
	if (!spin_trylock_irqsave(&rsp->fqslock, flags)) {
		rsp->n_force_qs_lh++; /* Inexact, can lose counts.  Tough! */
		return;	/* Someone else is already on the job. */
	}
	if (relaxed && (long)(rsp->jiffies_force_qs - jiffies) >= 0)
		goto unlock_ret; /* no emergency and done recently. */
	rsp->n_force_qs++;
	spin_lock(&rnp->lock);
	lastcomp = rsp->completed;
	signaled = rsp->signaled;
	rsp->jiffies_force_qs = jiffies + RCU_JIFFIES_TILL_FORCE_QS;
	if (lastcomp == rsp->gpnum) {
		rsp->n_force_qs_ngp++;
		spin_unlock(&rnp->lock);
		goto unlock_ret;  /* no GP in progress, time updated. */
	}
	spin_unlock(&rnp->lock);
	switch (signaled) {
	case RCU_GP_INIT:

		break; /* grace period still initializing, ignore. */

	case RCU_SAVE_DYNTICK:

		if (RCU_SIGNAL_INIT != RCU_SAVE_DYNTICK)
			break; /* So gcc recognizes the dead code. */

		/* Record dyntick-idle state. */
		if (rcu_process_dyntick(rsp, lastcomp, dyntick_save_progress_counter))
			goto unlock_ret;

		/* Update state, record completion counter. */
		spin_lock(&rnp->lock);
		if (lastcomp == rsp->completed) {

			rsp->signaled = RCU_FORCE_QS;
			dyntick_record_completed(rsp, lastcomp);
		}
		spin_unlock(&rnp->lock);
		break;

	case RCU_FORCE_QS:
```

Listing 5.5: Segment of `force_quiescent_state()` function in kernel v2.6.31.1. Code from file `kernel/rcutree.c`.

```c
bool add_gp(int x, int y)
{
        struct foo *p;

        p = calloc(1, sizeof(*p));
        if (!p)
                return -ENOMEM;
        spin_lock(&gp_lock);
        if (rcu_access_pointer(gp)) {
                spin_unlock(&gp_lock);
                return false;
        }
        p->a = x;
        p->b = x;
#ifdef ORDERING_BUG
        gp = p;
#else
        rcu_assign_pointer(gp, p);
#endif
        spin_unlock(&gp_lock);
        return true;
}

bool use_gp(void)
{
        struct foo *p;

        rcu_read_lock();
        p = rcu_dereference(gp);
        if (p) {
                BUG_ON(p->a != 42 || p->b != 42);
                /* do something with p->a, p->b */
                rcu_read_unlock();
                return true;
        }
        rcu_read_unlock();
        return false;
}

void *thread_publisher(void *arg)
{
        add_gp(42, 42);
        return NULL;
}

void *thread_subscriber(void *arg)
{
        use_gp();
        return NULL;
}
```

Listing 5.6: Test for validation of Tree RCU's Publish-Subscribe guarantee.

```
int r_x;
int r_y;

int x;
int y;

void *thread_reader(void *arg)
{
        set_cpu(cpu1);
        fake_acquire_cpu(get_cpu());

        rcu_read_lock();
        r_x = x;
        do_IRQ();
        r_y = y;
        rcu_read_unlock();
        cond_resched();
        do_IRQ();

        fake_release_cpu(get_cpu());
        return NULL;
}

void *thread_update(void *arg)
{
        set_cpu(cpu0);
        fake_acquire_cpu(get_cpu());

        x = 1;
        synchronize_rcu();
        y = 1;

        fake_release_cpu(get_cpu());
        return NULL;
}
```

Listing 5.7: Litmus test for validation of Tree RCU's Grace-Period guarantee.

Chapter 6

Concluding Remarks

In this thesis we modeled aspects of RCU and used the stateless model checking tool Nidhugg to validate the basic guarantees for both Tiny RCU and Tree RCU, taking the whole source code of these flavours as input.

More specifically, we described the basic kernel subsystems on which RCU relies, and some features of modern microprocessors and compilers that are crucial for RCU's operation. Additionally, we formulated the basic guarantee every RCU implementation needs to provide (Grace-Period guarantee), along with some implications imposed by today's compilers and microprocessors, and the way RCU works around those (Publish-Subscribe guarantee). Lastly, we created a test suite that checks whether these guarantees are satisfied or violated.

We validated the Grace-Period guarantee for both Tiny RCU and Tree RCU, under SC and TSO memory models. For Tree RCU, a similar test was ran for both v3.0 and v3.19 of the Linux kernel, with the test for v3.19 being much more important due to big differences in the implementation of these versions (e.g., the existence of the grace-period kthread). We also validated the Publish-Subscribe guarantee for Tree RCU under the TSO and the POWER memory models.

To show that our emulation of the kernel's environment is sound and to further strengthen our results, we reproduced a known kernel bug and also showed that a previously reported bug is not in reality a bug [LKMLb, LKMLc]. It is highly possible that the fix for the former bug also fixed the latter, since the fix for the former was submitted only a little time after the latter as an additional -though unrelated- patch [McKe09a, LKMLa].

Our work demonstrates that Nidhugg and stateless model checking can be used to verify *real* code from today's production systems with large codebases. In our tests we used the source code directly from four different versions of the Linux kernel; there were only minor changes regarding the usage of per-CPU variables, but these need not be performed manually – they can be scripted. The fact that Nidhugg was able to provide counterexamples for bugs in a few milliseconds or seconds demonstrates the strength of our approach.

Future Work

The work reported in this thesis can be extended in plenty of ways:

1. More sophisticated and refined tests for quiescent-state forcing could be added.

2. Different tests involving grace-period expediting could be added as well.

3. CPU hot plugs could be enabled.

4. Full-dynticks mode could be enabled.

5. Softirqs and interrupts could be modeled with separate threads.

6. Similar tests for preemptible kernels could also be added.

Bibliography

[Abdu14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson and Konstantinos Sagonas, “Optimal Dynamic Partial Order Reduction”, in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’14, pp. 373–384, New York, NY, USA, 2014, ACM.

[Abdu15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson and Konstantinos Sagonas, “Stateless Model Checking for TSO and PSO”, in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, pp. 353–367, New York, NY, USA, 2015, Springer-Verlag New York, Inc.

[Ahme15] Iftekhar Ahmed, Alex Groce, Carlos Jensen and Paul E. McKenney, “How Verified is My Code? Falsification-Driven Verification”, in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 737–748, November 2015.

[Algl13] Jade Alglave, Daniel Kroening and Michael Tautschnig, “Partial Orders for Efficient Bounded Model Checking of Concurrent Software”, in *Proceedings of the 25th International Conference on Computer Aided Verification*, pp. 141–157, 2013.

[AMDC] “Cool’n’Quiet”. Available: https://en.wikipedia.org/wiki/Cool%27n%27Quiet.

[Arca03] Andrea Arcangeli, Mingming Cao, Paul E. McKenney and Dipankar Sarma, “Using Read-Copy Update Techniques for System V IPC in the Linux 2.5 Kernel”, in *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, pp. 297–310, USENIX Association, June 2003.

[Beye15] Dirk Beyer, “Rules for 4th Intl. Competition on Software Verification”, in *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 4 2015. Available: https://sv-comp.sosy-lab.org/2015/rules.php.

[Bier03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman and Yunshan Zhu, “Bounded Model Checking”, *Advances in Computers*, vol. 58, 2003.

[Bove05] Daniel P. Bovet and Marco Cesati, *Understanding the Linux kernel*, O’ Reilly, 3rd edition, 2005.

[Call15] John Callaham, “Google says there are now 1.4 billion active Android devices worldwide”. Available: http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide, September 2015.

[Chri13] Maria Christakis, Alkis Gotovos and Konstantinos Sagonas, "Systematic Testing for Detecting Concurrency Errors in Erlang Programs", in *Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, pp. 154–163, Los Angeles, CA, USA, 2013, IEEE Computer Society.

[Clan] "LLVM Atomic Instructions and Concurrency Guide". Available: http://llvm.org/docs/Atomics.html#libcalls-atomic.

[Clar99] Edmund M. Clarke, Orna Grumberg, Marius Minea and Doron A. Peled, "State Space Reduction Using Partial Order Techniques", *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, 1999.

[Clar04] Edmund Clarke, Daniel Kroening and Flavio Lerda, "A tool for checking ANSI-C programs", in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176, Springer, 2004.

[Desn09] Mathieu Desnoyers, *Low-Impact Operating System Tracing*, Ph.D. thesis, Ecole Polytechnique de Montréal, December 2009. Available: http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf.

[Desn12] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais and Jonathan Walpole, "User-Level Implementations of Read-Copy Update", *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 2, pp. 375–382, February 2012.

[Desn13] Mathieu Desnoyers, Paul E. McKenney and Michel R. Dagenais, "Multi-core Systems Modeling for Formal Verification of Parallel Algorithms", *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 2, pp. 51–65, July 2013.

[Dijk] Edsger W. Dijkstra, "Over de sequentialiteit van procesbeschrijvingen". circulated privately.

[Dugg10] Abhinav Duggal, *Stopping Data Races Using Redflag*, Ph.D. thesis, Stony Brook University, 2010.

[Flan05] Cormac Flanagan and Patrice Godefroid, "Dynamic Partial-order Reduction for Model Checking Software", in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pp. 110–121, New York, NY, USA, 2005, ACM.

[GCCA] "Built-in Functions for Memory Model Aware Atomic Operations". Available: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html.

[Gode96] Patrice Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[Gode97] Patrice Godefroid, "Model checking for programming languages using VeriSoft", in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 147–186, 1997.

[Gode05] Patrice Godefroid, "Software Model Checking: The VeriSoft Approach", *Formal Methods in System Design*, vol. 26, no. 2, pp. 77–101, 2005.

[Gots13] Alexey Gotsman, Noam Rinetzky and Hongseok Yang, "Verifying Concurrent Memory Reclamation Algorithms with Grace", in *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, pp. 249–269, Berlin, Heidelberg, 2013, Springer-Verlag.

[Howe] David Howells, Paul E. McKenney, Will Deacon and Peter Zijlstra, "Linux Kernel Memory Barriers". Available: `https://www.kernel.org/doc/Documentation/memory-barriers.txt`.

[Inte] "Power Management States: P-States, C-States, and Package C-States". Available: `https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states`.

[Kerna] "Linux Kernel Documentation". Available: `https://www.kernel.org/doc/`.

[Kernb] "NO_HZ: Reducing Scheduling-Clock Ticks". Available: `https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt`.

[Kernc] "RCU Concepts". Available: `https://www.kernel.org/doc/Documentation/RCU/rcu.txt`.

[Kernd] "RCU Requirements". Available: `https://www.kernel.org/doc/Documentation/RCU/Design/Requirements/`.

[Linu] "The Linux kernel". `https://www.kernel.org/`.

[LKMLa] "rcu: clean up locking for `->completed` and `->gpnum` fields". `https://lkml.org/lkml/2009/10/30/212`.

[LKMLb] "rcu: fix long-grace-period race between forcing and initialization". `https://lkml.org/lkml/2009/10/28/196`.

[LKMLc] "rcu: Fix synchronization for `rcu_process_gp_end()` uses of `->completed` counter". `https://lkml.org/lkml/2009/11/4/69`.

[Love10] Robert Love, *Linux Kernel Development*, Addison-Wesley, 3rd edition, 2010.

[McKe] Paul E. McKenney, "RCU Linux Usage". Available: `http://www.rdrop.com/users/paulmck/RCU/linuxusage.html`.

[McKe98] Paul E. McKenney and John D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems", in *Parallel and Distributed Computing and Systems*, pp. 509–518, Las Vegas, NV, October 1998.

[McKe04] Paul E. McKenney, *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*, Ph.D. thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf`.

[McKe06] Paul E. McKenney, "Sleepable RCU". Available: `http://lwn.net/Articles/202847/` Revised: `http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf`, October 2006.

[McKe07a] Paul E. McKenney, "The design of preemptible read-copy-update". Available: `http://lwn.net/Articles/253651/`, October 2007.

[McKe07b] Paul E. McKenney, "Using Promela and Spin to verify parallel algorithms". Available: `http://lwn.net/Articles/243851/`, August 2007.

[McKe07c] Paul E. McKenney and Jonathan Walpole, "What is RCU, Fundamentally?". Available: `http://lwn.net/Articles/262464/`, December 2007.

[McKe08a] Paul E. McKenney, "Hierarchical RCU". Available: http://lwn.net/Articles/305782/, November 2008.

[McKe08b] Paul E. McKenney, "RCU part 3: the RCU API". Available: http://lwn.net/Articles/264090/, January 2008.

[McKe08c] Paul E. McKenney, "What is RCU? Part 2: Usage". Available:, http://lwn.net/Articles/263130/, January 2008.

[McKe09a] Paul E. McKenney, "Hunting Heisenbugs". Available: http://paulmck.livejournal.com/14639.html, 11 2009.

[McKe09b] Paul E. McKenney, "RCU: The Bloatwatch Edition". Available: http://lwn.net/Articles/323929/, March 2009.

[McKe10a] Paul E. McKenney, "Lockdep-RCU". Available: https://lwn.net/Articles/371986/, February 2010.

[McKe10b] Paul E. McKenney, "The RCU API, 2010 Edition". Available: http://lwn.net/Articles/418853/, December 2010.

[McKe13] Paul E. McKenney, "Structured Deferral: Synchronization via Procrastination", *ACM Queue*, vol. 11, no. 5, May 2013. Available: https://queue.acm.org/detail.cfm?id=2488549.

[McKe14] Paul E. McKenney, "The RCU API, 2014 Edition". Available: http://lwn.net/Articles/609904/, September 2014.

[McKe15a] Paul E. McKenney, "Requirements for RCU part 1: the fundamentals". Available: http://lwn.net/Articles/652156/, July 2015.

[McKe15b] Paul E. McKenney, "Requirements for RCU part 2 – parallelism and software engineering". Available: http://lwn.net/Articles/652677/, August 2015.

[McKe15c] Paul E. McKenney, "Requirements for RCU part 3". Available: http://lwn.net/Articles/652677/, August 2015.

[McKe15d] Paul E. McKenney, "Verification Challenge 4: Tiny RCU". Available: http://paulmck.livejournal.com/39343.html, 3 2015.

[Moln] Ingo Molnar and Arjan van de Ven, "Runtime locking correctness validator". Available: https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

[Musu08] Mandanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerald Basler, Piramanayagam Arumuga Nainar and Iulian Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs", in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 267–280, Berkeley, CA, USA, 2008, USENIX Association.

[Pele93] Doron Peled, "All from One, One for All: On Model Checking Using Representatives", in *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pp. 409–423, London, UK, UK, 1993, Springer-Verlag.

[Rela] "Relaxed-Memory Concurrency". Available: http://www.cl.cam.ac.uk/~pes20/weakmemory/.

[Seys12] Justin Seyster, *Runtime Verification of Kernel-Level Concurrency Using Compiler-Based Instrumentation*, Ph.D. thesis, Stony Brook University, 2012.

[Spar] "Sparse - a Semantic Parser for C". Available: `https://sparse.wiki.kernel.org/index.php/Main_Page`.

[Tass15] Joseph Tassarotti, Derek Dreyer and Viktor Vafeiadis, "Verifying Read-copy-update in a Logic for Weak Memory", in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pp. 110–120, New York, NY, USA, 2015, ACM.

[Trip09] Josh Triplett, Paul E. McKenney and Jonathan Walpole, "Relativistic Programming". Available: `http://wiki.cs.pdx.edu/rp/`, September 2009.

[Valm91] Antti Valmari, "Stubborn Sets for Reduced State Space Generation", in *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pp. 491–515, London, UK, UK, 1991, Springer-Verlag.

[Wikia] "Memory ordering". Available: `https://en.wikipedia.org/wiki/Memory_ordering`.

[Wikib] "Model checking". Available: `https://en.wikipedia.org/wiki/Model_checking`.

[Wikic] "Read-copy-update". Available: `https://en.wikipedia.org/wiki/Read-copy-update`.

Appendix A

Tree RCU Modified Functions

Below are listed some functions from `kernel/rcu/tree.c` file that have been modified for the bug injection procedure (see Section 5.5). The code relevant to the bug injections has been colored blue.

```
void rcu_sched_qs(void)
{
#ifdef LIVENESS_CHECK_2
        return;
#endif
        if (!rcu_sched_data[get_cpu()].passed_quiesce) {
                trace_rcu_grace_period(TPS("rcu_sched"),
                                       rcu_sched_data[get_cpu()].gpnum,
                                       TPS("cpuqs"));
                rcu_sched_data[get_cpu()].passed_quiesce = 1;
        }
}

static bool __note_gp_changes(struct rcu_state *rsp, struct rcu_node *rnp,
                            struct rcu_data *rdp)
{
        bool ret;

        /* Handle the ends of any preceding grace periods first. */
        if (rdp->completed == rnp->completed) {

                /* No grace period end, so just accelerate recent callbacks. */
                ret = rcu_accelerate_cbs(rsp, rnp, rdp);

        } else {

                /* Advance callbacks. */
                ret = rcu_advance_cbs(rsp, rnp, rdp);

                /* Remember that we saw this grace-period completion. */
                rdp->completed = rnp->completed;
                trace_rcu_grace_period(rsp->name, rdp->gpnum, TPS("cpuend"));
        }

        if (rdp->gpnum != rnp->gpnum) {
                /*
                 * If the current grace period is waiting for this CPU,
                 * set up to detect a quiescent state, otherwise don't
                 * go looking for one.
                 */
                rdp->gpnum = rnp->gpnum;
                trace_rcu_grace_period(rsp->name, rdp->gpnum, TPS("cpustart"));
                rdp->passed_quiesce = 0;
#ifdef LIVENESS_CHECK_1
                rdp->qs_pending = 0;
#else
```

```
                rdp->qs_pending = !!(rnp->qsmask & rdp->grpmask);
#endif
#ifdef FORCE_FAILURE_5
                rnp->qsmask &= ~rdp->grpmask;
#endif
                zero_cpu_stall_ticks(rdp);
        }
        return ret;
}

static int rcu_gp_init(struct rcu_state *rsp)
{
        struct rcu_data *rdp;
        struct rcu_node *rnp = rcu_get_root(rsp);

        rcu_bind_gp_kthread();
        raw_spin_lock_irq(&rnp->lock);
        smp_mb__after_unlock_lock();
        if (!ACCESS_ONCE(rsp->gp_flags)) {
                /* Spurious wakeup, tell caller to go back to sleep.  */
                raw_spin_unlock_irq(&rnp->lock);
                return 0;
        }
        ACCESS_ONCE(rsp->gp_flags) = 0; /* Clear all flags: New grace period. */

        if (WARN_ON_ONCE(rcu_gp_in_progress(rsp))) {
                /*
                 * Grace period already in progress, don’t start another.
                 * Not supposed to be able to happen.
                 */
                raw_spin_unlock_irq(&rnp->lock);
                return 0;
        }

        /* Advance to a new grace period and initialize state. */
        record_gp_stall_check_time(rsp);
        /* Record GP times before starting GP, hence smp_store_release(). */
        smp_store_release(&rsp->gpnum, rsp->gpnum + 1);
        trace_rcu_grace_period(rsp->name, rsp->gpnum, TPS(”start”));
        raw_spin_unlock_irq(&rnp->lock);

        /* Exclude any concurrent CPU-hotplug operations. */
        mutex_lock(&rsp->onoff_mutex);
        smp_mb__after_unlock_lock(); /* ->gpnum increment before GP! */

        /*
         * Set the quiescent-state-needed bits in all the rcu_node
         * structures for all currently online CPUs in breadth-first order,
         * starting from the root rcu_node structure, relying on the layout
         * of the tree within the rsp->node[] array.  Note that other CPUs
         * will access only the leaves of the hierarchy, thus seeing that no
         * grace period is in progress, at least until the corresponding
         * leaf node has been initialized.  In addition, we have excluded
         * CPU-hotplug operations.
         *
         * The grace period cannot complete until the initialization
         * process finishes, because this kthread handles both.
         */
        rcu_for_each_node_breadth_first(rsp, rnp) {
                raw_spin_lock_irq(&rnp->lock);
                smp_mb__after_unlock_lock();
                rdp = &rsp->rda[get_cpu()];
                rcu_preempt_check_blocked_tasks(rnp);
```

```
#ifdef FORCE_FAILURE_3
                rnp->qsmask &= ~rdp->grpmask;
#else
                rnp->qsmask = rnp->qsmaskinit;
#endif
                ACCESS_ONCE(rnp->gpnum) = rsp->gpnum;
                WARN_ON_ONCE(rnp->completed != rsp->completed);
                ACCESS_ONCE(rnp->completed) = rsp->completed;
                if (rnp == rdp->mynode)
                        (void)__note_gp_changes(rsp, rnp, rdp);
                rcu_preempt_boost_start_gp(rnp);
                trace_rcu_grace_period_init(rsp->name, rnp->gpnum,
                                            rnp->level, rnp->grplo,
                                            rnp->grphi, rnp->qsmask);
                raw_spin_unlock_irq(&rnp->lock);
                cond_resched_rcu_qs();
        }

        mutex_unlock(&rsp->onoff_mutex);
        return 1;
}

static void
rcu_report_qs_rnp(unsigned long mask, struct rcu_state *rsp,
                  struct rcu_node *rnp, unsigned long flags)
        __releases(rnp->lock)
{
        struct rcu_node *rnp_c;

#ifdef LIVENESS_CHECK_3
        return;
#endif
        /* Walk up the rcu_node hierarchy. */
        for (;;) {
                if (!(rnp->qsmask & mask)) {

                        /* Our bit has already been cleared, so done. */
                        raw_spin_unlock_irqrestore(&rnp->lock, flags);
                        return;
                }
                rnp->qsmask &= ~mask;
                trace_rcu_quiescent_state_report(rsp->name, rnp->gpnum,
                                                 mask, rnp->qsmask, rnp->level,
                                                 rnp->grplo, rnp->grphi,
                                                 !!rnp->gp_tasks);
#ifndef FORCE_FAILURE_6
                if (rnp->qsmask != 0 || rcu_preempt_blocked_readers_cgp(rnp)) {

                        /* Other bits still set at this level, so done. */
                        raw_spin_unlock_irqrestore(&rnp->lock, flags);
                        return;
                }
#endif
                mask = rnp->grpmask;
                if (rnp->parent == NULL) {

                        /* No more levels.  Exit loop holding root lock. */

                        break;
                }
                raw_spin_unlock_irqrestore(&rnp->lock, flags);
                rnp_c = rnp;
                rnp = rnp->parent;
                raw_spin_lock_irqsave(&rnp->lock, flags);
                smp_mb__after_unlock_lock();
                WARN_ON_ONCE(rnp_c->qsmask);
        }

        /*
         * Get here if we are the last CPU to pass through a quiescent
         * state for this grace period.  Invoke rcu_report_qs_rsp()
         * to clean up and start the next grace period if one is needed.
         */
        rcu_report_qs_rsp(rsp, flags); /* releases rnp->lock. */
}

static int __init rcu_spawn_gp_kthread(void)
{
        unsigned long flags;
        struct rcu_node *rnp;
        struct rcu_state *rsp;
        struct task_struct *t;

        rcu_scheduler_fully_active = 1;
#ifdef ENABLE_RCU_BH
        for_each_rcu_flavor(rsp) {
                t = kthread_run(rcu_gp_kthread, rsp, "%s", rsp->name);
                BUG_ON(IS_ERR(t));
                rnp = rcu_get_root(rsp);
                raw_spin_lock_irqsave(&rnp->lock, flags);
                rsp->gp_kthread = t;
                raw_spin_unlock_irqrestore(&rnp->lock, flags);
        }
#else
        t = kthread_run(rcu_gp_kthread, &rcu_sched_state, "%s",rcu_sched_state.name);
        rnp = rcu_get_root(&rcu_sched_state);
        raw_spin_lock_irqsave(&rnp->lock, flags);
        rcu_sched_state.gp_kthread = t;
        raw_spin_unlock_irqrestore(&rnp->lock, flags);
#endif
      rcu_spawn_nocb_kthreads();
      rcu_spawn_boost_kthreads();
      return 0;
}
early_initcall(rcu_spawn_gp_kthread);
```